VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rámec pro vytváření datových struktur v hlavní paměti pomocí C++

Main Memory Framework for a Data Structure Creation

2012 **Bc. Marek Rusnák**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Marek Rusnák**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Rámec pro vytváření datových struktur v hlavní paměti pomocí C++
Main Memory Framework for a Data Structure Creation

Zásady pro vypracování:

Součástí jazyka C++ není automatická správa paměti. Programátor tedy musí řešit uvolňování a vytváření objektů sám. Běhové prostředí pak má problémy s častým uvolňováním paměti, což může zpomalit celou aplikaci a navíc dochází k neefektivnímu nakládání s pamětí.

Součástí této práce je navržení rámce, který bude provádět automatickou správu paměti u jednoduchých datových struktur. Řešení bude mít jedno vlákno, jenž bude spravovat bloky paměti a přidělovat tuto paměť jednotlivým vláknům, jenž si o paměť zažádají. Datové struktury externího procesu by pak měly alokovat paměť z této vyhrazené paměti a v případě nutnosti požádat o další bloky paměti.

Řešení bude mít následující části:
1. Prozkoumání problémů běhových prostředí C++ při práci s hlavní pamětí.
2. Návrh a implementace procesu, který bude provádět správu bloků paměti.
3. Vytvoření několika základních datových struktur (pole, zásobník, fronta), jenž se budou schopny rychle alokovat z vyhrazené paměti.
4. Otestování celého řešení.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Radim Bača, Ph.D.**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave dňa 4.5.2012 .....

podpis

Chcem poďakovať svojmu konzultantovi
Ing. Radimovi Bačovi, Ph.D. za odborné konzultácie,
ako aj cenné rady a pripomienky pri písaní tejto práce.

Abstrakt

Diplomová práca sa zaoberá problémami alokácie a uvolňovania pamäte počas behu aplikácii. V prvej časti sa práca venuje teoretickému popisu pamäte z pohľadu fyzickej vrstvy, vrstvy operačného sytému a aplikačnej vrstvy. Podrobnejšie sú v práci rozobrané metódy prideľovania a uvolňovania pamäte v spoločnom kontexte s automatickou a manuálnou pamäťovou správou. Ďalšie kapitoly sa venujú základným dátovým štruktúram a možnostiam synchronizácie viacvláknových procesov v operačných systémoch MS Windows. Praktickou časťou diplomovej práce je návrh, implementácia a otestovanie rámca pre vytváranie dátových štruktúr v hlavnej pamäti pomocou C++. Rámec poskytuje rýchlu a efektívnu správu pamäte na aplikačnej úrovni, spoločne so základnými dátovými štruktúrami, ktoré sú schopné sa rýchlo alokovať z tejto pamäte.

Kľúčové slová: hlavná pamäť, halda, rámec, dátová štruktúra, pole, zásobník, fronta, zoznam, správa pamäte, správca haldy, blok pamäte, pamäťový pool, vlákno, C++

Abstract

The thesis deals with the problems associated with memory allocation in runtime applications. The first part is devoted to theoretical memory management in terms of physical layer, layer of the operating system and application layer. More work is discussed in the method of allocation and freeing of memory in the context of automatic and manual memory management. Other chapters deal with basic data structures, and the potential to synchronise multithreaded processes in MS Windows operating systems. The practical part of the thesis concerns project planning, implementation and testing of the main memory framework for creating a data structure in C++. The framework provides fast and efficient memory management at the application level, together with basic data structures that are able to quickly allocate blocks within this memory.

Key words: main memory, heap, framework, data structure, array, stack, queue, list, memory management, heap manager, memory block, memory pool, thread, C++

Zoznam skratiek

API – Application Programming Interface
CPU – Central Processing Unit
CRT – C Run-Time
DMA - Dynamic Memory Allocation
DRAM – Dynamická RAM
EEPROM – Electrically Erasable Programmable Read-Only Memory
EMM – Extended Memory Manager
EMS – Extended Memory Specification
GB – Giga Bajt
GUI – Graphical User Interface
HLA – High Level Assembly
KB – Kilo Bajt
LFH – Low Fragmentation Heap
MB – Mega Bajt
MOS – Metal Oxid Semiconductor
MS - Microsoft
MTBF – Mean Time Between Failures
OS – operačný systém
RAM – Random-Access Memory
ROM – Read-Only Memory
SRAM – Statická RAM
SSD – Solid State Drive
TB – Tera Bajt
TTL – Transistor-Transistor Logic
UMA – Upper Memory Area
XMM – eXtended Memory Manager
XMS – eXtended Memory Specification

Obsah

1 Úvod ... 3
2 Architektúra pamäte v počítači ... 5
2.1 Základná organizácia pamätí ... 5
2.2 Pamäť z pohľadu použitej technológie ... 6
2.3 Súčinnosť CPU s operačnou pamäťou ... 7
2.4 Logická organizácia hlavnej pamäte ... 8
3 Aplikačná úroveň správy pamäte ... 9
3.1 Typy premenných ... 9
3.2 Alokácia pamäte kompilátorom ... 9
3.2.1 Priestor programového kódu ... 10
3.2.2 Priestor pre dáta ... 10
3.2.3 Zásobník ... 10
3.3 Alokácia na halde ... 11
3.3.1 Volacie brány ... 13
3.3.2 Microsoft Windows a halda ... 13
3.4 Manuálna správa pamäte ... 16
3.5 Automatická správa pamäte ... 17
4 Metódy správy pamäte ... 18
4.1 Metódy prideľovania pamäte ... 18
4.1.1 Prideľovanie na zásobníku ... 18
4.1.2 Sekvenčné prideľovanie ... 19
4.1.3 Prideľovanie s obmedzenou veľkosťou bloku ... 21
4.1.4 Ostatné metódy prideľovania pamäte ... 22
4.2 Metódy automatického uvoľňovania pamäte ... 22
5 Viacvláknové procesy a ich synchronizácia v OS Windows ... 25
5.1 Kritická sekcia ... 25
5.2 Mutex ... 26
5.3 Semafór ... 26
5.4 Udalosť ... 26
5.5 Metered sekcia ... 26
6 Dátové štruktúry ... 28
6.1 Pole ... 28
6.2 Zásobník ... 28
6.3 Fronta ... 28
6.4 Zoznam ... 29
7 Analýza a návrh ... 30
7.1 Návrh správcu pamäte ... 30
7.2 Návrh dátových štruktúr rámca ... 35
7.3 Návrh testovacích prípadov ... 38

8 Implementácia 39
8.1 Použité technológie 39
8.1.1 Programovací jazyk C++ 39
8.1.2 Metered sekcie 39
8.2 Vlastná implementácia 39
8.2.1 Implementácia modulu správcu pamäte 40
8.2.2 Implementácia modulu dátových štruktúr rámca 40
8.2.3 Modul testov 43
9 Testovanie 45
9.1 Priebežné testy 45
9.2 Konečné testy 45
9.2.1 Výkonnostné testy 46
9.2.2 Test korektnosti 52
9.3 Problémy, na ktoré sme narazili pri testovaní 53
10 Záver 55
Zoznam použitej literatúry 56
Zoznam príloh

1 Úvod

Každý programátor vie, že ním vytvorené premenné a objekty, sa počas behu jeho aplikácie ukladajú do hlavnej pamäte. Máloktorý z nich však rieši otázku, či je spôsob alokácie a uvolňovania pamäte v danom programovacom jazyku skutočne efektívny a dostatočne rýchly. Za túto nevedomosť do istej miery môže vývoj a napredovanie vyšších programovacích jazykov, ktoré vo väčšine prípadov poskytujú medzi základnými službami i automatickú správu pamäte. Áno, nikto a samozrejme ani my sa nebránime inovatívnym riešeniam, ktoré nás odbremeňujú a uľahčujú nám prácu a nie je ani našim úmyslom spochybňovať prínos automatického uvolňovania pamäte. A čo viac, automatické uvolňovanie pamäte vyšších programovacích jazykov nám bezpochyby prácu uľahčuje. V niektorých situáciách sa však treba pozastaviť a zodpovedať si otázku, či nie je skutočne možné docieliť aby naša aplikácia pracovala rýchlejšie. Jednou z možností, ako toho docieliť je práve efektívnejšie riešenie správy pamäte. Nie je totiž ťažké dokázať a podložiť, že nevhodná alokácia a uvolňovanie pamäte vedie k zníženiu výkonu celej aplikácie a to bez ohľadu na konkrétny programovací jazyk. Trochu iný pohľad a prístup nám ponúka programovací jazyk C++, ktorý vo svojej štandardnej implementácii neposkytuje automatickú správu pamäte a programátor musí uvolňovať pamäť explicitne priamo v kóde programu. Manuálne uvolňovanie pamäte je určite rýchlejšie v porovnaní s uvolňovaním automatickým, neznamená to však že riešenie v C++ je ideálne. Častá alokácia a uvolňovanie pamäte hrá totiž významnú rolu pri znížení výkonu aplikácie aj u manuálneho uvolňovania v C++, kde zníženie výkonu plynie z jednoduchej skutočnosti, že predvolený správca pamäte je prirodzene určený pre všeobecné použitie a nerieši uvolňovanie pamäte najideálnejším spôsobom. Pokiaľ je teda v záujme programátora zvýšiť výkon svojich aplikácií a toto zvýšenie výkonu by chcel realizovať efektívnejšou správou pamäte, musí siahnuť po vhodnejšom riešení správy pamäte na aplikačnej úrovni, prípadne si musí správcu pamäte vytvoriť sám. Hlavnou činnosťou správy pamäte na aplikačnej úrovni je manipulácia s pamäťovými blokmi, ktoré sú predalokované počas inicializácie správcu a ukladané do tzv. *pamäťových poolov*. Pooly teda reprezentujú akýsi rezervoár voľných pamäťových blokov, ktoré sú následne prideľované externým procesom žiadajúcim o pamäť. Pamäťové pooly a správa pamäte na aplikačnej úrovni umožňujú rýchlejší prístup programu k pamäti, čo má za následok zrýchlenie celej aplikácie.

Diplomová práca sa zaoberá organizáciou pamäte v počítačoch z fyzickej úrovne, úrovne operačného systému a aplikačnej úrovne. Detailnejšie sa práca venuje manuálnej a automatickej správe pamäte a metódam prideľovania a uvolňovania pamäte. Praktickou úlohou práce je návrh, implementácia a otestovanie rámca pre vytváranie dátových štruktúr v hlavnej pamäti pomocou C++, ktorý bude prevádzať automatickú správu pamäte u jednoduchých dátových štruktúr. Vytvorený rámec by mal napomáhať programátorom pri vytváraní aplikácií v C++, ktoré budú s využitím rámca efektívnejšie pracovať s pamäťou, čo v konečnom dôsledku zvýši ich výkon.

Práca je rozdelená do 9 kapitol, kde prvá a poslená kapitola ja úvod a záver. Druhá a tretia kapitola sa zaoberajú teoretickou problematikou organizácie pamäte v počítačok z pohľadu troch úrovní: fyzickej úrovne, úrovne operačného systému a úrovne aplikačnej. Štvrtá kapitola sa venuje viacvláknovým procesom, konkrétnejšie synchronizácii vlákien v operačných systémoch Microsoft

Windows a piata kapitola v krátkosti popisuje základné dátové štruktúry. Praktickej časti práce sa venujú kapitoly šesť, sedem a osem, v ktorých je obsiahnutý základný algoritmus tvorby aplikácií. Šiesta kapitola sa zaoberá návrhom rámca pre vytváranie dátových štruktúr v hlavnej pamäti pomocou C++, v siedmej kapitole je následne popisaná implementácia a posledná kapitola popisuje otestovanie celého riešenia s výsledkami jednotlivých testov. Na priloženom CD sú k dispozícii zdrojové kódy rámca.

2 Architektúra pamäte v počítači

Na pojem počítačová pamäť sa môžme pozerať z troch úrovní: z úrovne technického vybavenia, z úrovne operačného systému a z úrovne aplikačnej [1]. Vo väčšine počítačových systémov sa tieto zložky do istej miery vyskytujú súčasne a vytvárajú vrstvy medzi užívateľským programom a konkrétnymi pamäťovými a riadiacimi prvkami. My sa budeme v prvej kapitole diplomovej práce v krátkosti venovať prvej a druhej úrovni. Popíšeme si základné rozdelenie a hierarchiu pamätí v počítači a bližšie sa pozrieme na hlavnú pamäť a jej logickú organizáciu. Aplikačnej úrovni, ktorá je pre nás a túto prácu najzaujímavejšia, sa budeme podrobnejšie venovať v 3. kapitole: Aplikačná úroveň správy pamäte.

2.1 Základná organizácia pamätí

Prítomnosť pamäte v počítači je pre jeho fungovanie nevyhnutnosťou. Mikroprocesor z nej číta programy, ktorými je riadený a zároveň do nej ukladá výsledky svojej práce. Pamätí je v počítači viac druhov. V zásade sa však dajú rozdeliť na primárne, s ktorými mikroprocesor bezprostredne pracuje a sekundárne, slúžiace pre dlhodobejšie ukladanie dát [2].

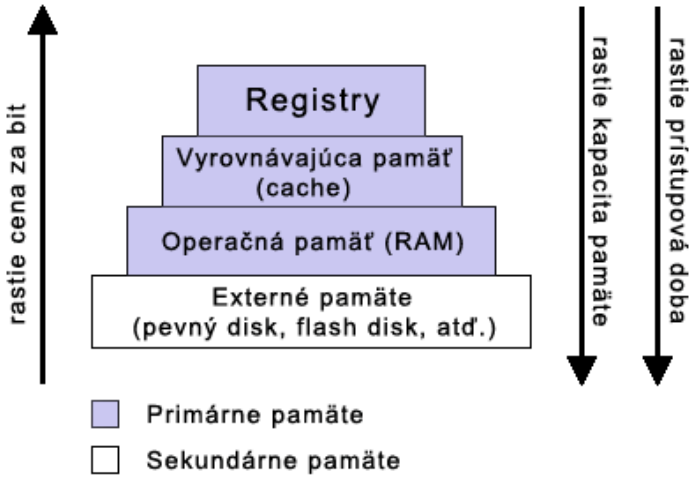
- *Primárne pamäte* – rozdeľujú sa do ďalších troch skupín:
  - *registre,*
  - *vyrovnávajúca (angl. cache) pamäť,*
  - *vnútorná (interná, operačná) pamäť.*
- *Sekundárne pamäte* – taktiež označované ako pamäť vonkajšia, alebo externá. Jedná sa o pamäť realizovanú väčšinou za pomoci zariadení používajúcich výmenné média v podobe bežných pevných diskov, FLASH (EEPROM) pamätí, alebo SSD diskov.

Ideálnym stavom pamäťovej architektúry počítača by bola prítomnosť len jedného, najrýchlejšieho typu pamäte v rámci celého počítača. Tento stav však v súčastnosti nie je možný, nakoľko pamäť s veľmi dobrými charakteristickými vlastnosťami je veľmi drahá a jej použitie v celom počítači by niekoľko násobne zdvihlo jeho cenu. Z tohto dôvodu vnikla postupom času určitá hierarchia pamätí, ktorá je zobrazená na prvom obrázku.

Registry
Na vrchole celej hierarchickej štruktúry pamätí sa nachádza veľmi rýchla pamäť tvorená pracovnými registrami umiestnenými priamo na mikroprocesore. Táto pamäť má zo všetkých typov najmenšiu kapacitu, približne 128 bytov, ale aj najkratšiu prístupovú dobu. Veľmi krátka prístupová doba je daná jednak charakteristickými vlastnosťami pamäte a jednak zapojením pracovných registrov priamo do ostatných štruktúr mikroprocesoru.

Vyrovnávajúca pamäť
Ďalším typom pamäte, ktorá sa nachádzajú na druhom mieste hierarchie, je vyrovnávajúca pamäť, tzv. cache. Vyrovnávajúca pamäť má dve úrovne a to L1 cache a L2 cache. L1 cache je umiestnená priamo na čipe mikroprocesoru a jej úlohou je zrýchlenie sprístupnenia dát pre mikroprocesor. Na druhej úrovni je o niečo pomalšia L2 cache, ktorá má však v porovnaní s L1 väčšiu kapacitu. Zvyčajne sa nachádza na samostatnom čipe umiestnenom na malom zásuvnom module spoločne

s mikroprocesorom a ďalšími podpornými obvodmi. To znamená, že i L2 cache je blízka mikroprocesoru a jej hlavnou úlohou je podpora a rozšírenie L1 cache pamäte. Z pohľadu výrobnej technológie, býva vyrovnávajúca pamäť založená buď na MOS SRAM, čiže statickej pamäti so šesťtranzistorovými pamäťovými bunkami, alebo na bipolárnej technológii. V dnešných PC vybavených výkonnejším procesorom a čipovou sadou sa vyskytujú už aj L3 cache pamäte, ktoré sú veľmi podobné pamäti L2 cache.



Obrázok 1. Hierarchia typov pamätí [3].

Vnútorná operačná pamäť

Až na tretej, respektíve štvrtej úrovni sa nachádza vnútorná operačná pamäť, ktorá je vo valnej väčšine prípadov osadená samostatne na základnej doske. Operačná pamäť je postavená na technológii polovodičových súčiastok, kde každý miniatúrny kondenzátor, fyzicky umiestnený do mriežky pamäťových buniek, predstavuje v nabitom stave jednotku a vo vybitom stave nulu. Do tohto typu pamäte sú zavádzané práve spustené programy, prípadne len ich nevyhnutná časť, a dáta s ktorými pracujú. Na rozdiel od pamätí s vyšších úrovní je operačná pamäť priamo, či nepriamo adresovateľná a teda plne prístupná programátorom.

Ďalšie úrovne reprezentujú už spomínané sekundárne pamäte, ktoré sú síce čiastočne súčasťou pamäťovej architektúry počítača, nebudeme sa im však v tejto práci podrobnejšie venovať. Za zmienku stojí pojem virtuálna pamäť, alebo virtuálny adresný priestor, ktorá sa všeobecné skladá z operačnej pamäte a z priestoru vyhradeného na pevnom disku. Operačný systém si pomáha pri práci s hlavnou operačnou pamäťou tým, že si dlhšiu dobu nepoužívané dáta presúva z hlavnej pamäte na virtuálnu pamäť pevného disku. Pri pokuse o prístup k tomuto bloku je daný blok opätovne načítaný do voľného miesta operačnej pamäte a sprístupnený programu. Tieto operácie sú transparentné z pohľadu aplikačného programátora, takže sa o vituálnu pamäť a jej adresovanie nemusí vôbec zaujímať [4].

2.2 Pamäť z pohľadu použitej technológie

Pamäte je možné rozdeliť z pohľadu použitej technológie do dvoch skupín: pamäte typu ROM a pamäte typu RAM. Pamäť typu ROM je užívateľsky i programátorsky transparentná a pre našu prácu nemá žiadny význam.

Pamäť typu RAM

S pamäťami typu RAM najčastejšie spolupracuje mikroprocesor. RAM pamäte sú v porovnaní s ROM rýchlejšie, energeticky závislé, umožňujú čítanie i zápis a v počítači je ich inštalovaná podstatne väčšia kapacita. Podľa kritéria, či sa jedná o RAM pamäť statickú, alebo dynamickú sa ďalej rozdeľujú na:

- *Statická RAM (SRAM)* – Pamäťová bunka je tvorená bistabilným klopným obvodom, čo je elektronický prvok nabývajúci dvoch stavov – 0, alebo 1. Výhodou tohto riešenia je nízka prístupová doba, ktorá sa pohybuje medzi 7,5 až 15 ns. Naopak, ich nevýhodou je väčšia zložitosť a z toho plynúce vyššie výrobné náklady. V súčasnosti sa pamäte SRAM používajú predovšetkým pre realizáciu pamätí typu cache, ktorých kapacita je samozrejme v porovnaní s operačnou pamäťou niekoľko násobne nižšia.
- *Dynamická RAM (DRAM)* – Pamäťová bunka je tvorená miniatúrnym kondenzátorom, ktorý predstavuje v nabitom stave jednotku a vo vybitom nulu. Miniatúrne kondenzátory majú malú kapacitu a tendenciu sa vybíjať aj v dobe, kedy je pamäť pripojená k zdroju elektrického napájania. Aby nedochádzalo k stráte informácie vybitím kondenzátorov, je potrebné kondenzátory periodicky dobíjať (tzv. refresh). Funkciu periodickej obnovy plnia niektoré z obvodov čipovej sady. Najčastejšie je týmto typom pamäte realizovaná hlavná operačná pamäť.

Tento typ pamäte prechádzal postupným vývojom a v súčastnosti existuje niekoľko typov pamätí RAM: SDRAM (Synchronous Dynamic RAM), DDR (Double Date Rate), DDR2, DDR3, DDR4, RDRAM (Rambus DRAM), FRAM atď.

2.3 Súčinnosť CPU s operačnou pamäťou

Centrálnym prvkom počítača je mikroprocesor – CPU, ktorého úlohou je spracovanie dát a interpretácia programových inštrukcií. Za účelom ukladania a získavania dát, výsledkov výpočtov a inštrukcií využíva mikroprocesor hlavnú operačnú pamäť, ktorá je s mikroprocesorom fyzicky prepojená pomocou adresovej (*angl.* address bus) a dátovej (*angl.* data bus) zbernice. Adresová a dátová zbernica umožňuje prenos adries a dát medzi mikroprocesorom a operačnou pamäťou.

Z pohľadu mikroprocesoru je operačná pamäť veľmi pomalou komponentou, ktorá značne spomaľuje jeho činnosť. Z tohto dôvodu je mikroprocesor vybavený vyrovnávajúcimi pamäťami L1, L2 až L3, ktorých úlohou je zrýchlenie prístupu mikroprocesoru k dátam uloženým v pomalej operačej pamäti. L1 pamäť načítava väčšie množstvo dát zo zbernice a dočasne ich uchováva pre potreby mikroprocesoru, ktorý nemusí čakať na pomalú odozvu zbernice, respektíve operačnej pamäte. L2 cache pracuje obdobným spôsobom ako jej predchodca a je umiestená medzi mikroprocesorom a hlavnou pamäťou. L2 cache je naviac ovládaná špeciálnym radičom, ktorý sa snaží predpovedať aké dáta bude mikroprocesor v najbližšej dobe požadovať. Občas môže dôjsť k situácii, že požadované dáta sa v L2 pamäti nenachádzajú a dáta musia byť vyžiadané priamo z operačnej pamäte. Napriek tomu je pravdepodobnosť nájdenia potrebných dát v L2 cache (tzv. Hit Rate) pomerne veľká, nakoľko mikroprocesor pracuje vždy určitú dobu v jednej oblasti operačnej pamäte. Posledným stupňom hierarchie vyrovnávajúcich pamätí je pamäť L3, ktorá je v porovnaní s L2 pomalšia, na druhú stranu však disponuje väčšou kapacitou.

2.4 Logická organizácia hlavnej pamäte

Hlavná pamäť slúži pre uloženie programu i samotných dát. Pamäťovým miestam sú priradené nezáporné celé čísla od *0, ... , N-1*, kde *N* je kapacita pamäte, ktorým sa hovorí adresa. Adresným priestorom je množina všetkých možných adries a býva určený počtom bitov v adrese.
Hlavná operačná pamäť je využívaná operačným systémom, užívateľom spustenými aplikáciami, BIOSmi hardvérových zariadení, ukladajú sa do nej I/O adresy atď. Je logické, že z dôvodu širokého spektra využitia operačnej pamäte a zachovania kompatibility medzi rôznymi operačnými systémami musí mať operačná pamäť jednotne definovanú štruktúru. Hlavná operačná pamäť je tak rozdelená do troch hlavných častí [5]:

- *konvenčná pamäť* (angl. Conventional memory),
- *rezervovaná pamäť* (angl. Reserved memory),
- *pamäť nad 1 MB.*

Konvenčná pamäť
Konvenčná pamäť zaberá prvých 640KB a rozdeľuje sa na oblasť vstupno/výstupných (I/O) adries a oblasť určenú pre prácu programov. Oblasť vstupno/výstupných adries má veľkosť 1KB a uchováva adresy umožňujúce komunikáciu mikroprocesoru s okolím. Programová oblasť začína od 1KB do 640KB a dnes už sa takmer nepoužíva.

Rezervovaná pamäť
Pamäť od 640KB do 1MB je rezervovaná pre technické prostriedky počítača a dnes je často označovaná ako UMA (*angl.* Upper Memory Area). UMA je rozšírením možnosti využitia rezervovanej pamäte, s ktorým prišla spoločnosť Microsoft v období, keď nízka kapacita operačnej pamäte bola problémovým miestom počítačovej architektúry.

Pamäť nad 1 MB
Pamäť nad 1 MB, tiež označovaná ako rozšírená pamäť, je tvorená zostávajúcou voľnou pamäťou. Jej veľkosť je daná pamäťovým modulom osadeným na základnej doske a operačným systémom. V dnešnej dobe je táto oblasť pamäte využívaná všetkými 32 a 64 bitovými aplikáciami. Prístup do tejto pamäťovej oblasti umožňujú správcovia pamäte (*angl.* memory managers), ktorý sú postavený na dvoch princípoch (stránkový, nestránkový) a podľa nich sa označuje aj rozšírená pamäť [5]:

- *Stránková pamäť (Expanded)* – historicky starší princíp, ktorý sa dnes už takmer nepoužíva. Princíp spočíva v tom, že sa pamäť rozdelí na stránky, do rezervovanej pamäte sa umiestni prepínač, ktorý posiela dáta na určité adresy určitých stránok. Prepínanie adries zabezpečujú stránkový manažéri EMS, alebo EMM.
- *Nestránková pamäť (Extended)* – jedná sa o novší, dnes plne využívaný princíp. Rozšírenie adresovej zbernice umožnilo vygenerovať viac adries a tak do tejto pamäte nie je potreba žiadnych prepínačov a stránok. Jedná sa teda o priame adresovanie a zabezpečujú ho pamäťový manažéri XMS, alebo XMM.

3 Aplikačná úroveň správy pamäte

V predchádzajúcej kapitole sme si popísali pamäť z pohľadu prvých dvoch úrovní, a to z úrovne technického vybavenia a čiastočne z úrovne operačného systému. V tejto kapitole sa bližšie pozrieme na aplikačnú vrstvu správy pamäte a popíšeme si niektoré najčastejšie používané metódy, respektíve algoritmy pre jej alokáciu a dealokáciu.
Aplikačnú vrstvu správy pamäte reprezentujú koncové uživateľské aplikácie, ktoré pristupujú a pracujú s pamäťou dvomi spôsobmi: alokácia na úrovni kompilátoru a alokácia na volnom úložisku tzv. halde [6]. Predtým, ako sa začneme venovať jednotlivým prístupom k pamäti, si vysvetlíme základné typy premenných, ich výhody a nevýhody z pohľadu životnosti a umiestnenia v pamäti.

3.1 Typy premenných

Premenné sa s pohľadu životnosti a umiestnenia rozdeľujú do troch skupín:

- *lokálne premenné,*
- *globálne premenné* (statické premenné, konštanty apod.),
- *premenné uložené na volnom uložisku.*

Všetky tri typy majú svoje výhody i nevýhody.

Výhodou lokálnych premenných je jednoduchosť správy pamäte, v ktorej sa premenné nachádzajú. Životnosť lokálnej premennej je limitovaná na konkrétnu funkciu, ktorá po ukončení svojej činnosti vracia riadenie hlavnému programu, a s týmto návratom zaniká i samotná lokálna premenná. Krátka životnosť lokálnych premenných je ich hlavnou nevýhodou, na druhú stranu však udržuje väčší „poriadok“ v pamäti i v samotnom kóde programu, s čím je úzko spojená i jeho bezpečnosť.

Globálne premenné síce riešia problém krátkej životnosti u lokálnych premenných, avšak za cenu neobmedzeného prístupu v rámci celého programu, čo môže byť v určitých prípadoch nebezpečné a nežiadúce.

Tretím typom sú premenné uložené na volnom úložisku, tzv. halde, ktoré rieši jednak problém krátkej životnosti, tak aj problém neobmedzeného prístupu. To, že volné úložisko eliminuje problémy lokálnych i globálnych premenných však neznamená, že inicializácia premenných na volnom úložisku je najideálnejším a bezproblémovým riešení. Jeho hlavnou nevýhodou (v niektorých prípadoch môže byť i výhodou) je uvoľňovanie pamäte, ktoré môže byť automatické, alebo manuálne. Automatické uvoľňovanie pamäte prebieha síce automaticky, má však väčšiu časovú i pamäťovú réžiu, ktorá sa premieta aj do behu samotného programu. A naopak, manuálny spôsob je síce rýchlejší, avšak uvoľňovanie pamäte je plne v rukách programátora, kedy môže samozrejme dochádzať k väčšej chybovosti. Programátor si tak určuje, kedy potrebnú pamäť alokuje a vyhradená pamäť ostáva programu k dispozícii, až do chvíle jej explicitného uvolnenia.

3.2 Alokácia pamäte kompilátorom

Prvým prístupom k pamäti je alokácia samotným kompilátorom. Podpora tohto prístup k pamäti je odlišná u rôznych programovacích jazykov a prekladačov a je plne závislá na konkrétnom

vývojovom prostredí. Pamäť alokovaná kompilátorom je ďalej štrukturovaná. Najčastejšie sa stretneme s následujúcimi oblasťami, pričom jedna aplikácia môže obsahovať niekoľko oblastí určených pre kód programu a niekoľko oblastí nazývaných zásobník:

- *priestor programového kódu*,
- *priestor pre dáta*,
- *zásobník*,
- *volné úložisko – halda*.

3.2.1 Priestor programového kódu

Priestor programového kódu obsahuje inštrukcie nachádzajúce sa v HLA (angl. High Level Assembly), programe, kde HLA je assembler s podporou príkazov vyšších programovacích jazykov (napr. if, while, for). HLA prekladá každú strojovú inštrukciu príkazu na jedno, alebo viac bytovú hodnotu, ktoré CPU po spustení programu interpretuje, ako strojové inštrukcie na základe ukazateľov inštrukcií zo sady registrov. HLA v podstate spája programové inštrukcie zo systémom, ktorý týmto spôsobom získava informáciu, že spustený program môže vykonávať dané inštrukcie a môže čítať dáta z priestoru kódu. Priestor programového kódu slúži len pre čítanie (READ ONLY), čo znamená že z aplikácie nemôžme žiadnym spôsobom vykonávať zápis do tejto pamäťovej oblasti.

V poslednom odstavci sme spomenuli pojem registre, ktoré sa používajú pre interné funkcie správy behu programu, sledovanie vrcholu zásobníku a ukazovateľa na aktuálnu inštrukciu. Nachádzajú sa, respektíve sú zabudované priamo do CPU.

3.2.2 Priestor pre dáta

Priestor pre dáta sa často označuje ako statická pamäť, z čoho vyplýva že oblasť má pevnú veľkosť a existuje počas celého behu programu. Priestor pre dáta sa môže ďalej členiť na oblasť statickú, read-only oblasť a oblasť globálnych premenných. Z poslednej vety vyplýva, že priestor pre dáta je určený pre ukladanie a uchovávanie globálnych premenných, statických premenných, konštánt atď., ktoré sú dostupné počas celého behu programu [7].

3.2.3 Zásobník

Zásobník je špeciálna oblasť pamäte, ktorá je alokovaná pre spustený program a uchovávajú sa v ňom dáta, ktoré požaduje každá funkcia programu. V zásobníku sa teda uchovávajú lokálne premenné, ktorých životnosť je limitovaná funkciou v ktorej sa nachádzajú. Zásobník je dátová štruktúra fronty typu LIFO - posledný dovnútra, prvý von, ktorej sa budeme podrobnejšie venovať v 5. kapitole. Po zavolaní každej programovej funkcie dochádza v zásobníku k následujúcim činnostiam:

1. do zásobníku sa uloží návratová adresa a vstupné parametre funkcie
2. behom života funkcie sa lokálne premenné postupne pridávajú do zásobníku
3. pri návrate z funkcie sa všetky premenné postupným odobraním zo zásobníku odstránia [8].

V niektorých prípadoch kompilátor alokuje pamäť aj na volnom úložisku, tzv. halde. Nakoľko je halda primárne určená pre dynamické prideľovanie pamäte počas behu programu, budeme sa jej podrobnejšie venovať v následujúcej kapitole.

3.3 Alokácia na halde

Halda (*angl.* heap), tiež označovaná ako volné úložisko, alebo dynamicky alokovaná pamäť (*angl.* Dynamic Memory Allocation - DMA), slúži k dynamickej alokácii a uvolňovaniu objektov používaných programom. Halda využíva časť pamäte mimo pamäť vyhradenú k alokácii kompilátorom a líši sa od nej vo všetkých smeroch. Na rozdiel od pamäte alokovanej kompilátorom je veľkosť a životnosť pamäte pridelenej z haldy takmer nepredvídateľná, nakoľko sa obsadenie haldy dynamicky mení za behu programu. Aby bolo dynamické prideľovanie a uvolňovanie pamäte na halde efektívne, vyžaduje samostatného agenta, tzv. správcu haldy (*angl.* heap manager). Správca haldy so sebou prináša isté výhody, ale aj vyššiu pamäťovú i časovú réžiu vyvolanú ďalším programovým kódom potrebným pre jeho činnosť. Správca haldy je teda ďalším rozdielom oproti pamäti alokovanej kompilátorom, kde priestor pre dáta nevyžaduje žiadnu správu a správa zásobníku je primárne limitovaná na začiatok a koniec funkcií.

Rovnako, ako zásobník s lokálnymi premennými, alebo priestor pre dáta so svojimi globálnymi premennými, aj halda má svoje výhody a nevýhody. Medzi hlavné výhody haldy patrí:

- *Životnosť* – alokovanú pamäť na halde je možné ponechať aktívnu a odkaz na vyalokovanú pamäť predávať medzi ostatnými funkciami a triednymi inštanciami. To znamená, že s pamäťou môžeme pracovať na akomkoľvek mieste, čo je zásadný rozdiel oproti lokálnym premenným.
- *Veľkosť* – veľkosť voľnej pamäte na halde je niekoľko násobne väčšia oproti prvému typu alokácie kompilátorom a jej kontrolu je možné realizovať s väčšou presnosťou.

Medzi nevýhody haldy patrí:

- *Vyššia pracnosť* – alokácia a uvolňovanie pamäte musí byť riešená explicitne v kóde programu, čo zvyšuje pracnosť.
- *Vyššia chybovosť* – explicitná alokácia a uvolňovanie pamäte na halde má vyššie riziko vzniku chyby, kedy programátor zabudne uvoľniť alokovaný blok pamäte a v pamäti vznikajú tzv. diery, alebo *zblúdilé ukazovatele* (*angl.* dangling pointers) [9].

Práca a prístup k pamäti na halde sa značne opiera o knižnice užívateľského módu (*angl.* user mode), ktorých obsahom sú funkcie určené pre manipuláciu s pamäťovými blokmi a môžu byť volané, buď priamo v kóde programu, alebo nepriamo virtuálnym strojom. Napríklad v jazyku C má programátor k dispozícii funkcie `malloc()` a `free()` deklarované v knižnici `stdlib.h`. V C++ môže s haldou pracovať pomocou príkazov `new` a `delete`.

Ako sme už spomínali, u programovacích jazykov využívajúcich haldu, sa pamäť na halde prideľuje výhradne dynamicky. Pre toto dynamické prideľovanie existuje niekoľko metód, ktoré sa od seba líšia jednak efektivitou využitia pamäte, a jednak veľkosťou časovej a pamäťovej réžie. Prideľovanie a uvolňovanie pamäte, či už manuálne, alebo automatické, so sebou prináša určité obmedzenia, s ktorými je treba počítať a zohľadňovať ich pri návrhu, implementácii a použití v konkrétnych situáciách. Medzi hlavné faktory, alebo obmedzenia patrí:

- *časová réžia* – čas spotrebovaný správou pamäte počas behu programu,
- *pamäťová réžia* – pamäť spotrebovaná pre administráciu správy pamäte. Pamäťová réžia v sebe zahŕňa i nevhodné „zaokrúhľovanie“ veľkosti pamäťových blokov a s tým spojený vznik fragmentácie.

Správa pamäte nie je jednoduchou záležitosťou a jej nesprávna implementácia v nízkoúrovňových i vysokoúrovňových programovacích jazykoch, môže generovať množstvo problémov. Tieto problémové miesta a ich nevhodné riešenie môžu značne ovplyvniť róbustnosť a rýchlosť aplikácie a to bez ohľadu na to, či sa jedná o manuálnu, alebo automatickú správu pamäte. U manuálnej správy je hlavným problémom ľudský faktor a jeho pochybenie, kedy môže dochádzať k úniku pamäte a k jej predčasnému uvolňovaniu. A naopak, automatická správa síce rieši nedostatky manuálneho uvolňovania pomocou zložitejších algoritmov, má však oveľa vyššiu réžiu. Nie je totiž úplne jednoduché rozhodnúť, či má algoritmus v danom momente blok pamäte uvolniť, alebo ponechať. Správca haldy samozrejme nemôže spracovať a prideliť každej požiadavke pamäť s rovnakou efektivitou. Všeobecne môže u manuálnej i automatickej správy dochádzať k nasledujúcim chybám a nedostatkom v efektivite nakladania s pamäťou na halde [1].

Chyby vyskytujúce sa v správe pamäte:

- *Predčasné uvolnenie pamäte (angl. dangling pointers)* – problém vyskytujúci sa najčastejšie u manuálnej správy pamäte, kedy sa snaží program pristupovať k pamäti, ktorá už bola pred nejakým časom uvolnená. V programe tak ostávajú stále platné odkazy tzv. dangling pointers,
- *Únik pamäte (angl. memory leak)* – k úniku pamäte dochádza v situáciu kedy program alokuje neustále novú pamäť, bez uvolňovania starej, nepotrebnej a nepoužívanej pamäťe. V tomto prípade dochádza k postupnému znižovaniu voľnej pamäte na halde až k pádu programu,
- *Externá fragmentácia* – externou fragmentáciou sa označuje stav, kedy nie je možné prideliť blok pamäte požadovanej veľkosti, aj napriek tomu že je celková hodnota voľnej pamäte väčšia. Dochádza k tomu vtedy, keď nesprávne navrhnutý algoritmus alokácie rozdelí počas behu programu voľnú pamäť postupne na rovnaké, veľmi malé bloky. Malé, voľné bloky sa striedajú s obsadenými blokmi a v extrémnom prípade môže dôjsť k situácii, kedy správca haldy nemá k dispozícii súvislý blok požadovanej veľkosti.

Nedostatky vyplývajúce z neefektívnej správy pamäte:

- *Interná fragmentácia* – k vnútornej fragmentácii dochádza vtedy, ak správca haldy pridelí a vráti blok pamäte, ktorého veľkosť je väčšia oproti požadovanej. Z aplikácie napríklad prichádza požiadavka na blok veľkosti 128B a správca haldy uvolní a vráti blok veľkosti 512B. Zbytkových 384B pamäte zostáva nevyužitých.
- *Nesprávna lokalita odkazov* – prístupy k pamäti sú rýchlejšie pokiaľ pracujeme s blokmi pamäte, ktoré nie sú od seba príliš vzdialené. Pokiaľ správca haldy umiestni bloky, ku ktorým program pristupuje súčasne, ďaleko od seba, môže to viesť k zníženiu výkonu programu.

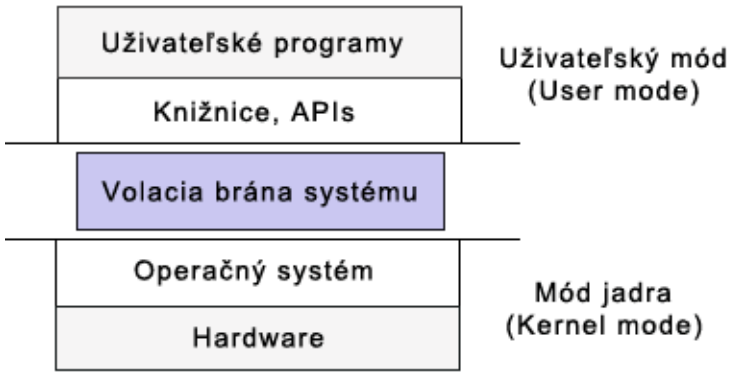
- *Neprispôsobivý návrh* – metóda správy pamäte môže dopredu počítať s určitými vlastnosťami programu, ako je napríklad veľkosť požadovaných pamäťových blokov, životnosť prideľovaných objektov atď. Nesprávne navrhnutá prispôsobivosť algoritmu k zmenám, môže taktiež zvýšiť celkovú réžiu správy pamäte [1].

Pred tým, ako sa dostaneme k jednotlivým metódam alokácie a uvolňovania pamäte, si popíšeme ako aplikačné knižnice komunikujú s operačným systémom, a bližšie sa pozrieť na riešenie haldy v OS Windows.

3.3.1 Volacie brány

Každá uživateľská aplikácia musí nejakým spôsobom komunikovať s operačným systémom, aby bola schopná minimálne pracovať s operačnou pamäťou. Komunikáciu zabezpečuje tzv. *volacia brána systému* (*angl.* system call gate), ktorá vytvára rozhranie medzi uživateľským módom (*angl.* user mode) a módom jadra (*angl.* kernel mode) pomocou ktorého uživateľská aplikácia pristupuje k funkciám a službám operačného systému (obrázok 2).

Operačné systémy Windows disponujú jadrom typu mikrokernel, ktoré obsahuje len základné funkcie systému a ostatné funkcie a služby operačného systému sa nachádzajú mimo jadra v pridružených aplikáciách. Rozhranie sa v OS Windows označuje ako *Windows API* a umožňuje spoločný prístup k obidvom skupinám funkcií. Počas implementácie uživateľských aplikácií tak nie je nutné rozlišovať medzi funkciou knižnice a využívaním služby operačného systému [10].



Obrázok 2: Volacia brána medzi uživateľským módom a módom jadra.

3.3.2 Microsoft Windows a halda

Pojmom správa pamäte je v operačných systémoch Windows označovaná takmer všetka činnosť spojená s pamäťou počítača z poľadu uživateľských aplikácii a operačného systému. O správu pamäte sa stará správca pamäte, ktorého štruktúra je dosť zložitá a jeho podrobný popis by spokojne vyšiel na samostatnú prácu. Napriek jeho komplikovanosti a širokej škále poskytovaných služieb však jeho základnými úlohami naďalej zostávajú [11]:

- *Preklad/Mapovanie virtuálneho adresového priestoru procesu do fyzickej pamäte*
- *Stránkovanie dočasne nevyužívaného obsahu pamäte na disk, v situáciách keď dôjde k jej kapacitnému vyčerpaniu*

Správca pamäte poskytuje sadu systémových služieb ako je alokácia a uvoľňovanie virtuálnej pamäte, zdieľanie pamäte medzi procesmi, mapovanie súborov do pamäte, prevádzanie virtuálnych stránok na disk, obnovovanie informácii ohľadom rozsahu virtuálnych stránok, zmeny v ochrane virtuálnych stránok, uzamykanie virtuálnych stránok v hlavnej pamäti apod. Väčšina služieb je dostupná prostredníctvom rozhrania Windows API, ktoré za účelom ich volania obsahuje tri skupiny funkcii [11]:

- *funkcie stránkovej granularity virtuálnej pamäte* (*angl.* page granularity virtual memory function),
- *funkcie mapovania súborov do pamäte* (*angl.* memory-maped file functions),
- *funkcie haldy* (*angl.* heap functions).

Niektoré z uvedených služieb poskytuje správca pamäte i vlastným komponentám módu jadra systému. Samostatnú časť potom tvoria tzv. *haldy módu jadra* (angl. kernel-mode heaps), ktoré správca pamäte alokuje počas zavádzania operačného systému. Tieto haldy sú po ich alokácii dostupné výhradne pre potreby operačného systému a správca pamäte s nimi ďalej nepracuje. Haldy módu jadra sa vyskytujú v dvoch režimoch, ako stránkované pooly (*angl.* paged pools) a nestránkované pooly (*angl.* nonpaged pools).

Virtuálny pamäťový priestor je teda počas závadzania operačného systému správcom pamäte rozdelený na systémovú oblasť a aplikačnú oblasť, pričom len aplikačná oblasť zostáva naďalej pod jeho správou [12].

Správca pamäte pri svojej činnosti zarovnáva každú oblasť procesom rezervovanej pamäte tak, aby každý pridelený *systémový blok* začínal na jednotnom rozhraní definovanom hodnotou *systémovej alokačnej granularity* (*angl.* allocation granularity). Jej veľkosť je 64KB a je jednotná pre všetky operačné systémy Windows. OS Windows sa jednoducho snaží všetky systémové bloky pamäte alokované, či už na halde, alebo mimo haldu zarovnávať do 64KB blokov [13]. U malých objektov s veľkosťou pod 64KB, však alokačná granularita znižuje výkon a využitie pamäte, čo je značne neefektívne hlavne u haldy, kde sú často ukladané objekty menších veľkostí. Tento problém rieši *správca haldy* (*angl.* heap manager), ktorého úlohou je adresácia objektov na halde prostredníctvom svojich vlastných hodnôt granularity, čím sa potlačí nežiadúca 64KB alokačná granularita správcu pamäte. Správca haldy je teda samostanou komponentou správy pamäte, ktorá je logicky umiestnená medzí virtuálnym pamäťovým priestorom a správcom pamäte a jeho činnosťou vznikajú *bloky správcu haldy* s veľkosťou menšou ako 64KB.

Alokačná granularita správcu haldy v operačných systémoch Windows je relatívne malá. Pre 32 bitový operačný systém je hodnota 8B a pre 64 bitový operačný systém 16B. Správca haldy sa v OS Windows nachádza v `NTDLL.DLL` a `NTOSKRNL.EXE` a medzi jeho najpoužívanejšie funkcie patrí:

- *HeapCreate a HeapDestroy* – vytvára a odstraňuje jednotlivé haldy,
- *HeapAlloc* – alokuje blok na halde,
- *HeapFree* – uvoľňuje blok na halde,
- *HeapReAlloc* – zmení veľkosť existujúcej alokácie,

- *HeapLock a HeapUnlock* – kontrolujú spoločný prístup k blokom haldy,
- *HeapWalk* – počíta položky a oblasti na halde.

C a C++ behové prostredie (CRT) využíva správcu haldy pri prevádzaní funkcií *malloc* a *free*, respektíve *new* a *delete*.

Ďalšou zaujímavosťou je, že v OS Windows môžu procesy bežiace v pamäti počítača využívať dva typy háld. Každý proces má minimálne jednu, tzv. vychádzajúcu procesnú haldu (*angl.* default process heap) a jednu, alebo niekoľko súkromných háld. Defaultná halda sa vytvorí pri spustení procesu a je aktívna a prítomná počas jeho celého behu. Jej vychádzajúca minimálna veľkosť je 1MB, pričom je možné počiatočnú veľkosť nastaviť špecifikáciou parametru počiatočnej veľkosti. Samozrejmosťou je jej dynamické zväčšovanie počas behu aplikácie, na základe potrieb individuálnych procesov. Defaultná halda môže byť explicitne využívaná programom, alebo implicitne využívaná internými funkciami OS Windows. Súkromné haldy sú vo vyhradnom užití konkrétneho procesu, ktorý ich vytvára a spravuje pomocou hore uvedených funkcií (*HeapCreate, HeapDestroy* atď.).

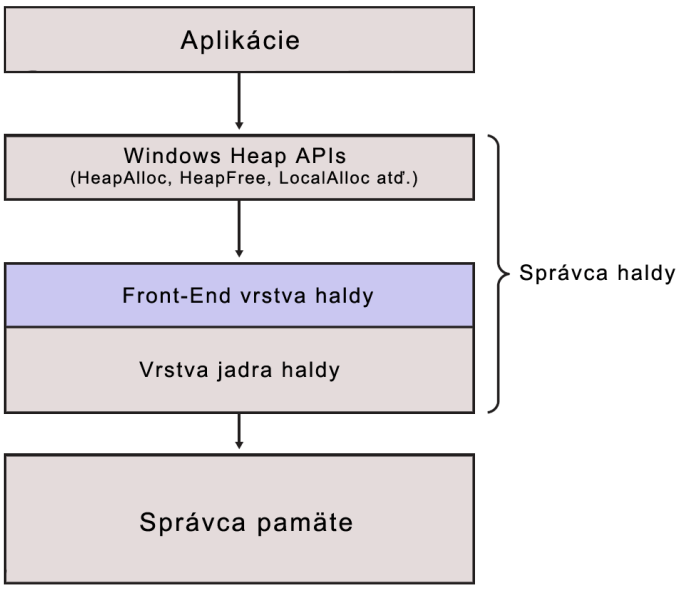
V OS Windows je štruktúra správcu haldy rozdelená do dvoch vrstiev:

- *front-end vrstva* (nepovinná),
- *jadro haldy.*

Jadro haldy spravuje základnú funkcionalitu, ktorá je spoločná pre implementáciu uživateľského módu i módu jadra a medzi jeho základné činnosti patrí: správa blokov haldy vo vnútri systémových blokov, správa systémových blokov, pravidlá pre rozšírenie haldy, správa veľkých blokov haldy a pod.

Haldy uživateľského módu môžu obsahovať, mimo jadra haldy, aj tzv. front-end vrstvu, ktorej úlohou je zníženie fragmentácie haldy. V súčasnosti existujú dva typy, respektíve dve možnosti realizácie front-end vrstvy a to tzv. *look-aside listy* a tzv. *halda s malou fragmentáciou* (*angl.* Low Fragmentation Heap - LFH), pričom každá halda môže v jednom čase používať len jednu front-end vrstvu. Look-aside listy sa na úrovni front-end vrstvy vyskytovali u starších operačných systémov a dnes ich nájdeme len v haldách módu jadra. Od verzie Windows XP je front-end vrstva správcu haldy realizovaná výhradne s LFH.

LFH je algoritmus založený na metóde prideľovania pamäte best fit, ktorého úlohou je efektívnejšie nakladanie s pamäťovými blokmi správcu haldy. Bližšie sa budeme venovať metóde best-fit v kapitole 5, kde sa pozrieme aj na algoritmus LFH.



Obrázok 3: Vrstvy správcu haldy [11].

3.4 Manuálna správa pamäte

Manuálna správa pamäte vyžaduje, aby programátor explicitne alokoval i uvoľňoval pamäť na halde. U manuálnej správy je teda plná zodpovednosť za alokáciu a uvoľňovanie pamäte na programátorovi, ktorý prostredníctvom funkcie `malloc()` v C, alebo príkazu `new` v C++, vyalokuje pamäť, ktorá je bežiacemu programu k dispozícii až do chvíle jej explicitného uvolnenia pomocou funkcie `free()` v C, alebo príkazu `delete` v C++. Výsledkom explicitnej správy pamäte sú jednoduchšie algoritmy behové prostredia, ktoré znižujú časovú i pamäťovú réžiu potrebnú pre správu pamäte. Na druhú stranu, manuálna správa pamäte so sebou prináša väčšiu chybovosť, nakoľko je v hre ľudský faktor. Medzi dve najhlavnejšie a najčastejšie chyby u manuálnej správy pamäte patrí: predčasné uvolnenie pamäte a únik pamäte. Pri nepozornosti a nesprávnej dealokácii zostáva pamäť obsadená a pre operačný systém i samotný program nedostupná a nepoužiteľná. V krajných prípadoch môže nesprávna manipulácia s pamäťou na halde viesť k extrémnemu nárastu spotrebovanej pamäte, až k pádu programu. Nevýhodou manuálnej správy je taktiež veľká pracnosť pre programátora, ktorý musí explicitne uvolňovať všetku alokovanú pamäť na halde. Manuálna správa pamäte má samozrejme aj svoje výhody ako je:

- nižšia časová i pamäťová réžia potrebná pre správu haldy a s tým spojená vyššia rýchlosť behu aplikácie,
- lepšia predstava programátora o alokovanej a uvolnenej pamäti.

Manuálna správa pamäte je v súčasnosti rýchlejšia a má menšiu pamäťovú réžiu v porovnaní s automatickou správou pamäte. Posledné štúdie však ukazujú, že moderné a prepracované alogoritmy automatickej správy sa neustále zrýchľujú a zefektívňujú, a rýchlosť uvolňovania pamäťových blokov u automatickej správy sa takmer vyrovnáva manuálnemu uvolňovaniu [14].

Aj keď je v súčasnosti manuálna správa pamäte rýchlejšia oproti správe automatickej, tak to neznamená že explicitné uvolňovanie pamäte má nulovú časovú réžiu. Uvolňovanie pamäte

pomocou *delete* (prípadne *free()* v C) nie je úplne jednoduchým mechanizmom, nakoľko proces spustený príkazom *delete* musí často previesť zložitejšie úlohy, ako len uvolnenie pamäte spojené s objektom. Tento proces sa označuje ako finalizácia a zahŕňa v sebe činnosti ako: uzatvorenie súborov, samostatnú finalizáciu doplnkových objektov apod. Pri finalizácii môže taktiež dochádzať k tzv. „kríženiu" pamäte, ktorá nebola dlhšiu dobu využívaná a jej bloky sa tak môžu nachádzať v rôznych oblastiach pamäte, čo spôsobuje problematickejší prístup k týmto blokom a spomalenie celého procesu uvolňovania. Tieto problémy samozrejme komplikujú uvolňovanie pamäte nie len v manuálnej správe, ale aj v automatickej, realizovanej tzv. *garbage collectorom*. Podrobnejšie sa k metódam uvolňovania pamäte dostaneme v 4. kapitole.

3.5 Automatická správa pamäte

Automatická správa pamäte nazývaná, ako zberač odpadu (*angl.* Garbage Collector) poskytuje komplexné riešenie uvolňovania pamäte na halde, ktorá je dlhšiu dobu programom nepoužívaná a pre program nepotrebná. V súčasnosti je automatická správa pamäte obľúbená a garbage collector je jednou zo základných služieb ponúkaných vysokoúrovňovými jazykmi, ako je napríklad Java, .NET apod. Garbage Collector totiž rieši bežné problémy manuálnej správy, ako je únik a predčasné uvolnenie pamäte, odbremeňuje programátora od explicitného uvolňovania, čo sú určite jasné znaky prínosu a obľúbenosti tohto mechanizmu. Na druhú stranu, za prínos garbage collectoru platíme celkovým spomalením aplikácie a vyššou pamäťovou réžiou, ktorá je potrebná pre programový kód garbage collectoru a ukladanie informácií o objektoch v pamäti. Aby bol totiž garbage collector schopný rozhodnutia, či je daný objekt v pamäti dostupný, alebo nedostupný, musí mať o tomto stave nejakú informáciu, čo generuje nadbytočné dáta ktoré nie sú potrebné pre beh samotného programu. Problémom je aj spotreba procesorového času garbage collectorom, kedy môže dochádzať až k značným pauzám v behu programu. Všetko je to však závislé na správnej voľbe metódy uvolňovnia pamäte.

Základom mechanizmu automatického uvolňovania pamäte je odhalenie „živosti", respektíve dosiahnuteľnosti pamäte z istej sady programových premenných, ako globálnych, lokálnych premenných a registrov, a na ktoré sa nie je možné dostať pomocou ukazovateľov. Postupom času vznikali mnohé algoritmy založené na počítaní referencií, vyhľadávaní objektov, rozdelení programu do generácií apod., ktorých zámerom bolo aj je zefektívnenie funkcie garbage collectoru. O tejto problematike viac v kapitole 4.

Medzi automatickou a manuálnou správou pamäte nie je možné zvoliť metódu lepšiu, efektívnejšiu alebo výhodnejšiu, nakoľko sú výhody a nevýhody jednotlivých typov úplne odlišné a kontraproduktívne. Samozrejme by sme mohli za účelom rozhodnutia previesť pokus, podobne ako previedol a popísal Bill Blunden vo svojej knihe Memory Managment: Algorithms and Implementation in C/C++, ktorý bez akéhokoľvek polemizovania preukázal výhody manuálnej správy pamäte z pohľadu rýchlosti a pamäťovej réžie [6]. Na opačnej strane však stojí garbage collector a jeho takmer bezchybná dealokácia pamäte, ktorá jednoznačne „prebíja" problémovú explicitnú dealokáciu a jej chybovosť u manuálnej správy pamäte. Existujú teda situácie vhodné pre využitie manuálnej správy pamäte a rovnako existujú situácie pre použitie správy automatickej.

4 Metódy správy pamäte

Obsahom správy pamäte je súbor metód, pomocou ktorých operačný systém prideľuje a regeneruje, čiže uvoľňuje pamäť. Kolekciu metód dostupných v správcoch haldy je možné rozdeliť do dvoch skupín: metódy prideľovania pamäte a metódy automatického uvoľňovania pamäte, pričom metódy prideľovania pamäte sa týkajú automatickej i manuálnej správy a metódy uvoľňovanie len správy automatickej (Garbage Collector). V tejto kapitole si postupne ukážeme základné a najčastejšie používané algoritmy obidvoch skupín.

4.1 Metódy prideľovania pamäte

Prideľovanie pamäte je jedným zo základných činností správcu haldy a jej efektivita má veľký vplyv na rýchlosť behu aplikácie a fragmentáciu haldy. Pri prideľovaní pamäte môže dochádzať k rôznym situáciám, ktoré plynú od postupnosti požiadaviek na alokáciu a dealkoáciu pamäťových blokov, obsadenia pamäte, fragmentácie pamäte apod. Pokiaľ by bol postup alokácie a uvoľňovania blokov pamäte postavený na pevnom mechanizme podobnom zásobníku (posledný alokovaný blok je prvý uvolnený), alebo fronte (prvý alokovaný blok je prvý uvolnený), bola by správa pamäte veľmi jednoduchá. Častejšie však dochádza k situáciám kedy sú bloky pamäte rôznej veľkosti alokované a uvoľňované v ľubovoľnom poradí. Aj z tohto dôvodu sa prideľovanie pamäte na halde často označuje, ako dynamické prideľovanie, nakoľko pamäťové bloky nie sú alokoujú sekvenčne, ale dynamicky.

Náhodná postupnosť alokácií a dealokácií pamäťových blokov teda spôsobuje nežiadúcu fragmentáciu pamäte, či už externú alebo internú. Externá fragmentácia zo sebou prináša väčšie riziká a nevýhody spojené s nemožnosťou alokácie väčšieho bloku pamäte, aj napriek tomu že pamäť obsahuje dostatok voľného miesta. Z tohto dôvodu je hlavným cieľom metód prideľovania pamäte maximálna eliminácia externej fragmentácie, pre ktorú sú v niektorých prípadoch schopné obetovať a pripustiť aj vznik fragmentácie internej. Niektoré metódy prideľovania pamäte radšej alokujú väčší blok pamäte aký bol požadovaný, aby predišli možnému vzniku externej fragmentácie. V následujúcom texte si vymenujeme a v krátkosti popíšeme základné metódy prideľovania pamäte, ktoré je možné rozdeliť do štyroch skupín:

- *prideľovanie na zásobníku*,
- *sekvenčné prideľovanie* (*angl.* sequencial-fit):
    - *first fit*,
    - *circular fit*,
    - *best fit*,
    - *worst fit,*
- *prideľovanie s obmedzenou veľkosťou bloku* – tzv. Buddy metódy, alebo Buddy systém,
- ostatné metódy prideľovania pamäte.

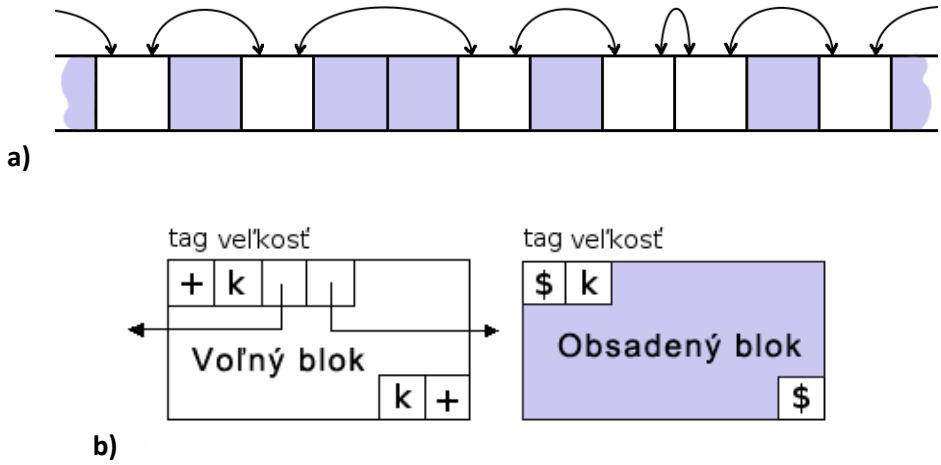
4.1.1 Prideľovanie na zásobníku

Jedná sa o najjednoduchšiu metódu prideľovania pamäte z jediného súvislého pamäťového bloku, organizovaného ako zásobník. Pri jej realizácii musí mať správca pamäte k dispozícii len adresu

začiatku a konca voľnej pamäte. Každej požiadavke je priradené aktuálna adresa začiatku voľnej pamäte a táto adresa je následne zvýšená o veľkosť požiadavky, pričom sa kontroluje prekročenie limitu voľnej pamäte. U tejto metódy spomenieme aj uvoľňovanie pamäte, ktoré môže byť implementované dvomi spôsobmi. Buď je metóda pre uvoľňovanie implementovaná, ako prázdna metóda, čiže pri jej volaní nedochádza k žiadnemu uvoľňovaniu pamäte, alebo je možné bloky pamäte uvoľňovať v opačnom poradí, ako boli pridelené. Metóda prideľovania na zásobníku je veľmi rýchla, samostatné využitie však má len u lokálnych premenných, prípadne vo funkcii tzv. subalokátoru pre prideľovanie pamäte v rámci bloku získaného inou metódou [1].

4.1.2 Sekvenčné prideľovanie

Ďalšou metódou prideľovania pamäte je *sekvenčné prideľovanie* (*angl.* sequencial-fit). Sekvenčné metódy sa snažia nájsť blok pamäte, v závislosti na type sekvenčného prideľovania, ktorý bude vyhovovať požiadavke. Metódy sekvenčného prístupu sú postavené na organizácii voľných pamäťových blokov do obojsmerného spojového zoznamu (obrázok 4a), ktorý umožňuje rýchlejší pohyb medzi voľnými pamäťovými blokmi. Obojsmerný spojový zoznam sa ozančuje, ako *zoznam prázdnych blokov*.



Obrázok 4: Obojsmerný spojový zoznam volných pamäťových blokov (sivé bloky – obsadená pamäť, biele bloky – voľná pamäť). Na obrázku b ja znázornená réžia správcu haldy v jednotlivých pamäťových blokoch.

Na obrázku 4a je znázornený spomínaný obojsmerný spojový zoznam, ktorého uzly sú tvorené z voľných pamäťových blokov. Aby bolo možné realizovať spojený zoznam musí si správca pamäte „zapožičať“ malý priestor v každom z týchto voľných blokov, kam uloží ukazovatele na predchádzajúci a následujúci blok, informáciu o veľkosti bloku a tzv. *tag bit* označujúci začiatok a koniec voľného bloku. Tagy, spoločne s informáciou o veľkosti bloku, sa nachádzajú na začiatku i konci pamäťového bloku, pričom na začiatku bloku je umiestnený počiatočný tag a na konci tag koncový (obrázok 4b). Pri pridelení pamäťového bloku sa blok odtrhne zo zoznamu prázdnych blokov, tým stráca odkazy na predchádzajúci a následujúci blok a ostáva mu len informácia o začiatku a konci bloku a informácia o svojej veľkosti. V prípade požiadavky na pridelenie pamäte určitej veľkosti *m*, musí správca pamäte samozrejme počítať s tým, že jeho réžia zaberá nejakú časť bloku a vybrať s pamäťového poolu dostatočne veľký blok.

Alokácia pamäťového bloku sekvenčným prideľovaním: Nech *m* je veľkosť požadovanej pamäte a *n* je pamäťová réžia sekvenčného prideľovania u každého bloku. Nájdi voľný blok (závisí na metóde – first fit, best fit atď.) a priraď ho žiadajúcemu procesu.

1. Nájdi voľný blok minimálnej veľkosti *k*, tak aby $k \geq (m+n)$.
2. Ak je $k > (m+n)$, rozdeľ blok veľkosti *k* na dva bloky o veľkosti $(m+n)$ a $(m+n)-k$. Vráť blok veľkosti $(m+n)-k$ späť do pamäťového poolu.
3. Vráť prvý voľný blok veľkosti *m* ako odpoveď na požiadavku

Pri prideľovaní pamäte, tak môžu nastať dve situácie. V prvom prípade sa veľkosť *k* pridelovaného bloku pamäte rovná požadovanej veľkosti *m* a pamäťový blok sa kompletne uvolní z pamäťového poolu. V druhom prípade je veľkosť *k* pridelovaného bloku pamäte väčšia oproti veľkosti *m*, ktorá bola požadovaná a algoritmus musí byť trochu chytrejší a zložitejší. Časť bloku požadovanej veľkosti *m* sa pridelí žiadateľovi a zo zbytkovej nevyužitej časti bloku sa vytvorí nový prázdny blok, ktorý ostáva v pamäťovom poole, respektíve v zozname prázdnych blokov.

Spojový zoznam spoločne s informáciami uloženými v pamäťových blokoch slúži správcovi, aby čo narýchlejšie poskytol blok pamäte požadovanej veľkosti. V prípade požiadavky tak správca pamäte prechádza spojový zoznam a hľadá vhodný blok pre pridelenie.

Návrat bloku do pamäťového poolu po jeho uvolnený sa v najjednoduchšom prípade vybaví ukazovateľmi na predchádzajúci i následujúci prázdny blok a zmenia sa informatívne tagy na hodnotu definujúcu, že blok je od tejto chvíle prázdny. Existujú samozrejme pokročilejšie algoritmy, ktoré efektívnejšie pracujú s pamäťovými blokmi pri ich návrate do pamäťového poolu, ako je napríklad zlúčenie susedných blokov, čím sa získajú väčšie pamäťové bloky a potláča sa tým do istej miery externá fragmentácia. Na metódy uvolňovania sa bližšie pozrieme v ďalšej podkapitole.

Ako sme už spomínali, medzi základné techniky sekvenčného prideľovania patrí: first fit, circural fit, best fit, worst fit [1].

First fit – Metóda výberu prvého voľného bloku prechádza zoznam voľných blokov a vyberie z neho prvý blok, ktorého veľkosť *k* je väčšia, alebo rovnaká požadovanej veľkosti. V prípade, že je blok väčší, zostávajúca časť je vložená späť do zoznamu, ako nový voľný blok.

Circular fit – Jedná sa o jednoduchú modifikáciu metódy first fit často označovanú, ako next fit. U metódy circular fit si algoritmus pri hľadaní voľného bloku pamätá miesto, kde skončil a pri ďalšom hľadaní začína hľadať na pozícii, kde predchádzajúce hľadanie skončilo. Algoritmus tak nemusí prehľadávať vždy začiatok zoznamu, ktorý sa pri metóde first fit aj tak veľmi rýchlo vyalokuje a fragmentuje, ale môže efektívnejšie a rýchlejšie pracovať cez celý zoznam pamäťového poolu.

Best fit – Metóda best fit je reakciou na nevýhody predchádzajúcich metód, ktoré vytvárajú vo veľkej miere externú fragmentáciu pamäte. Princípom tejto metódy je vyhľadanie najmenšieho voľného bloku pamäte, ktorého veľkosť je väčšia alebo rovná požadovanej. Tým sa do určitej miery zabráni zbytočnému „plytvaniu“ veľkými blokmi, ktoré bude správca pamäte potrebovať neskôr. Best fit tak vedie k minimalizácii strát zaistením menšej externej fragmentácie pamäte a výrazne spomaľuje aj jej vznik. Metóda má však dve nevýhody. Prvou nevýhodou je, že algoritmus pri prehľadávaní zoznamu s prázdnymi pamäťovými blokmi musí prejsť celým

zoznamom, čo je určite časovo náročné a spomaľuje to celý algoritmus. Druhým problémom je vznik veľmi malých, ďalej nepoužiteľných fragmentov pamäte. Vylepšením best fit metódy je tzv. *segregovaná pamäť*, kde sú bloky približne rovnakej veľkosti umiestené v rôznych pamäťových pooloch a prehľadávanie redukovaného poolu zrýchľuje celý algorimus. V implementácii segregovanej pamäte sa tak nachádza niekoľko zoznamov s voľnými blokmi, rozdelených na základe ich veľkosti, podobne ako je tomu u metódy prideľovania s obmedzenou veľkosťou bloku (kapitola 4.1.3).

Worst fit – Metóda worst fit funguje úplne opačne, ako best fit. Worst fit totiž vyhľadáva, čo najväčší blok, ktorý pridelí a zbytok vráti do pamäťového poolu. Metóda tak spôsobuje veľkú externú fragmentáciu a algoritmus musí opäť prehľadávať celý zoznam prázdnych blokov, rovnako ako u metódy best fit. Jediným vhodným použitím tejto metódy by bola situácia keby všetky požiadavky o pamäť mali rovnakú veľkosť, čo je takmer nemožné.

LFH

Halda s malou fragmentáciou - LFH (*angl.* Low Fragmentation Heap) je algoritmus front-end vrstvy správcu haldy v OS Windows. LFH vychádza z metódy best-fit a jej hlavnou úlohou je redukcia internej fragmentácie udržením malej pamäťovej „stopy“ každého procesu. Množstvo aplikácii bežiacich v OS Windows totiž využíva len veľmi malú časť pamäte z haldy, rádovo pod 1MB, čo vyžaduje efektívny algoritmus pre nakladanie s pamäťovými blokmi a tým je práve LFH.

LFH bráni fragmentácii správou alokovaných blokov správcu haldy, v predom stanovený rozsah veľkosti bloku (*angl.* bucket). V prípade, že proces alokuje pamäť z haldy, LFH vyberie bucket, ktorý mapuje najmenšiu veľkosť bloku, ktorá bude dostatočne veľká pre požadovanú veľkosť. Najmenší alokovaný blok je 8B a prináleží prvému bucketu, ktorý alokuje pamäť od 1B do 8B, následuje druhý bucket s alokáciou medzi 9 až 16B atď. (prehľad bucketov, ich granularita a rozsah sa nachádza v tabuľke prílohy A.1). Správca haldy obsahuje aj ďalšie algoritmy, ktoré zefektívňujú manipuláciu s pamäťou na halde. Jedným z nich je napríklad algoritmus, ktorým je správca haldy schopný uchovávať informáciu o najčastejšie používanej veľkosti bucketu u daného procesu. To znamená, že správca automaticky siaha po „top“ buckete s daným rozsahom, čo do istej miery zvyšuje výkon v prideľovaní pamäťových blokov [11].

4.1.3 Prideľovanie s obmedzenou veľkosťou bloku

Medzi hlavné nevýhody sekvenčného prideľovania pamäte patrí pomalé vyhľadávanie voľných blokov s časovou zložitosťou $\Theta(n)$ v najhoršom prípade a pamäťová réžia, ktorú potrebuje správca pamäte pre prácu s voľnými, alebo obsadenými pamäťovými blokmi. Rovnako je u sekvenčného prideľovania problematickejšie aj zlučovanie susedných pamäťových blokov. Prideľovanie pamäte s obmedzenou veľkosťou bloku, tzv. *buddy systém* rieši problémy sekvenčného prideľovania, pomocou hierarchického delenia voľného pamäťového priestoru na časti. Buddy systém zo sebou prináša efektívnejšie hľadanie pamäťového bloku správnej veľkosti, jednoduchšie zlúčenie susedných blokov a pamäťová réžia správcu je nižšia nakoľko buddy systém nevyžaduje žiadne informatívne značky vo vnútri pamäťových blokov.

U buddy systému sa teda jedná o hierarchické rozdelenie voľnej pamäte. Pamäť je rozdelená do jednotlivých úrovní podľa veľkosti pamäťového bloku s pravidlom, že na vyššej úrovni sa vždy nachádzajú bloky s väčšou veľkosťou. Na každej úrovni tak vzniká spojový zoznam obsahujúci bloky rovnakej veľkosti. V najjednoduchšom prípade je pamäť rozdelená do dvoch

častí, čiže do dvoch zoznamov a v najhoršom prípadne na 2^N zoznamov, kde N je celé číslo. Pre rozdeľovanie pamäte existujú dve varianty: *Binárne prideľovanie* a *Fibonacciho pridelovanie*.

U binárneho prideľovania buddy systém predpokladá, že pamäť je veľkosti 2^N, pričom voľné i obsadené bloky budú vždy veľkosti 2^k pre $k <= N$ a v danej chvíli sa môžu v pamäti nachádzať voľné i obsadené bloky rôznej veľkosti.

Alokácia pamäťového bloku buddy systémom: Nech m je veľkosť požadovanej pamäte. Nájdi prázdny blok minimálnej veľkosti m a priraď ho žiadajúcemu procesu.

1. Ak m je menšie ako veľkosť najmenšieho pamäťového bloku hierarchie, nastav m ako najmenšie.
2. Nájdi najmenšiu možnú hodnotu k, takú aby $2^k \geq m$.
3. Ak neexistuje voľný blok 2^k, tak rekurzívne alokuj blok veľkosti 2^{k+1}, rozdeľ blok na dva voľné bloky veľkosti 2^k a posuň bloky o úroveň nižšie.
4. Vráť prvý voľný blok veľkosti 2^k ako odpoveď na požiadavku

Fibonacciho varianta znižuje vnútornú fragmentáciu využitím kompaktnejšej sady možných veľkostí bloku. Vzhľadom k tomu, že každý prvok Fibonacciho postupnosti je súčtom dvoch predchádzajúcich prvkov, je možné blok vždy bez zbytku rozdeliť na dva bloky, ktorých veľkosti sú opäť prvkami postupnosti.

Nevýhodou buddy systému je vznik internej fragmentácie, nakoľko je pamäť delená na bloky veľkosti postupných mocnín 2. Napríklad pre požadovanú veľkosť 257 buddy systém alokuje blok veľkosti 512. Naopak medzi jeho výhody patrí: menšia externá fragmentácia, rýchlejšie vyhľadávanie voľných pamäťových blokov a jednoduchšie spájenie susedných blokov. Hlavným cieľom tohto systému je, aby bolo možné pri uvolnení bloku nájsť jeho suseda jednoduchým adresovým výpočtom [15].

4.1.4 Ostatné metódy prideľovania pamäte

Okrem spomínaných metód existuje množstvo ďalších ad-hoc prístupov k správe pamäte. Jedným z ďalších možnosti je rozdelenie pamäte do niekoľkých zón, kde správa pamäte každej zóny využíva inú metódu. Na základe charakteru a veľkosti požiadavky sa vyberie zóna, z ktorej bude pamäť pridelená. Ďalším prístupom je zavedenie štandardnej veľkosti pre všetky požiadavky o pamäť atď.

My budeme v praktickej časti realizovať správcu pamäte, ktorý prenesie všetku zodpovednosť za pamäť do aplikačnej úrovne. Rámec bude pracovať s dopredu vyalokovanou pamäťou, ktorú bude poskytovať okolitým procesom a o ktorej pridelenie si budú žiadať prostredníctvom rozhrania rámca.

4.2 Metódy automatického uvolňovania pamäte

Počas behu programu nastáva veľmi často situácia, kedy je alokovaná pamäť pre program už zbytočná a je potrebné ju uvolniť za účelom ďalšieho použitia, či už programom alebo operačným systémom. K tomu nám samozrejme slúžia algoritmy pre uvolňovanie pamäte, ktorým sa budeme venovať v tejto kapitole.

V predchádzajúcich kapitolách sme si rozdelili správu pamäte na automatickú a manuálnu. Toto delenie vychádza hlavne z mechanizmu uvolňovania pamäte, kedy je uvoľovanie v plnej moci programátora u manuálnej správy, alebo prebieha automaticky na pozadí u správy automatickej. Prideľovanie pamäťových blokov je automatické v obidvoch prípadoch a najčastejšie realizované vyššie uvedenými metódami prideľovania pamäte.

Manuálne uvolňovanie, alebo regenerácia pamäte prebieha na pokyn programátora, ktorý manuálne uvolní blok pamäte reprezentovaný priamou adresou v pamäti. Manuálne uvolňovanie teda nie je nijak zložité a nevyžaduje ani zložité algoritmy, ktoré by museli zásadne riešiť efektivitu uvolňovania.

Automatické uvolňovanie pamäti sa dnes bežne označuje ako Garbage Collection a časť správy pamäte, ktorá má uvolňovanie na starosti sa označuje ako Garbage Collector. V dnešnej dobe existuje veľké množstvo rôznych algoritmov realizujúcich uvolňovanie pamäte, v zásade sú však všetky postavené na dvoch základných princípoch: *metódy založené na sledovaní odkazov* a *metódy založené na čítačoch odkazov*. Špeciálnou skupinou sú potom *metódy inkrementálne*, ktoré uvolňujú pamät po častiach a uvolňovanie sa strieda s prevádzaním vlastného programu. Medzi najznámejšie algoritmy automatického uvolňovania pamäte patrí [16]:

- *Algoritmus počítania referencií* – u algoritmu počítania referencií je každému objektu, pri jeho vytvorení, priradený tzv. *čítač referencií* s počiatočnou hodnotou nastavenou na 1. Každá nová referencia zvýši jeho hodnotu o jedna a každá dereferencia hodnotu zníži. Vo chvíli, keď hodnota čítača dosiahne nulu je objekt pre program nepotrebný a je možné pamäť vyhradenú pre objekt uvolniť.
- *Mark-sweep algoritmus* – mark and sweep algoritmus, tzv. *metóda dvojfázového značkovania* je postavená na dosiahnuteľnosti objektov z koreňového objektu. Objekty sú prechádzané v dvoch fázach, kde v prvej fáze (*angl.* mark phase) sú objekty označené a v druhej fáze (*angl.* sweep phase) sú neoznačené, čiže nedosiahnuteľné objekty z pamäte uvolnené.
- *Mark-compact algoritmus* – mark-compact je algoritmus, ktorý taktiež pracuje v dvoch fázach. V prvej fáze dochádza k značkovaniu dosiahnuteľných objektov, rovnako ako u algoritmu mark-sweep. V druhej fáze sú označené bloky zjednotené (*angl.* compacted) a skopírované do inej časti pamäte, kde vytvoria súvislú nefragmentovanú pamäťovú oblasť. Druhá časť pamäte je použitá pre nové prideľovanie pamäte.
- *Uvolňovanie s kopírovaním* – algoritmus uvolňovania kopírovaním je často označovaný ako *kopírovací collector* (*angl.* copying collector). Jeho hlavným princípom je rozdelenie priestoru haldy na niekoľko častí. Ak jedna z častí dosiahne plného stavu, dosiahnuteľné objekty sa prekopírujú do novej oblasti. Algoritmus opäť prebieha v dvoch fázach, podobne ako predchádzajúce algoritmy.
- *Algoritmus založený na oblastiach* – algoritmus založený na oblastiach (*angl.* region-based) rozdeľuje haldu na niekoľko oblastí, kde každá oblasť je spravovaná samostatne, nezávisle na oblasti druhej.

Garbage collectory je možné rozdeliť do ďalších štyroch kategórii, podľa kritéria ako a kedy k uvolňovaniu pamäte dochádza. U tohto delenia len v krátkosti spomenieme o čo sa konkrétne jedná [16]:

- *Stop-the-world algoritmy* – algoritmy tohto typu uvolňujú pamäť bez prerušenia. To znamená, že algoritmus musí prebehnúť celý a počas jeho behu nemôže samotný program prevádzať žiadnu činnosť.
- *Paralelné algoritmy* – paralelný algoritmus beží rovnako, ako v prvom prípade bez prerušenia, s tým rozdielom že do učitej miery rieši redukciu časovej réžie potrebnej na uvolnenie pamäte. Uvolňovanie pamäte je v tomto prípade rozdelené do niekoľkých, paralelne bežiacich vlákien, ktoré sú medzi sebou synchronizované.
- *Inkrementálne algoritmy* – algoritmy uvolňujú pamäť priebežne v krátkych impulzoch. Počas týchto krátkych imulzov, v ktorých dochádza k parciálnemu uvolňovaniu pamäte, je program pozastavený. Krátke impulzy však nespôsobujú až tak dlhé časové oneskorenia, ako napríklad stop-the-world algoritmy
- *Súčinné (angl. concurrent) algoritmy* – súčinné algoritmy bežia vo svojom vlastnom vlákne, paralelne s vlastným programom. Nedochádza teda k pozastaveniu hlavnej aplikácie, algoritmy sú teda vhodné pre real-time alebo interaktívne aplikácie. V tomto prípade musí byť precízne riešená synchronizácia a konkurenčný prístup k objektom uloženým v pamäti.

5 Viacvláknové procesy a ich synchronizácia v OS Windows

V tejto kapitole sa nebudeme venovať presnej činnosti a implemenácii vlákien a viacvláknových aplikácií, ale zameriame sa konkrétnejšie na synchronizáciu vlákien v operačných systémoch Windows, pre ktoré je určený aj námi implementovaný rámec.

Pred tým, ako sa dostaneme k možnostiam samotnej synchronizácie, by bolo vhodné si vysvetliť dva pojmy, ktoré sa často zamieňajú, alebo splývajú v pojem jeden. Jedná sa o pojmy proces a vlákno:

- *Proces* – proces je v systéme chápaný, ako súhrn kódu programu, dát programu, zásobníku, údajom o procesom otvorených súboroch a taktiež informácie ohľadom spracovania signálov.
- *Vlákno* – vlákna sú časti programu, dokonca ich môžme označiť za funkcie programu, ktoré sú prevádzané súčasne. Existujú v rámci jedného procesu a väčšinou zdieľajú jeho prostriedky uvedené v definícii procesu.

Pod jedným procesom tak môže bežať jedno, alebo niekoľko vlákien, kde každé vlákno má svoj vlastný zásobník a svoje vlastné kópie registrov CPU. Ostatné prostriedky, ako napríklad súbory, statické dáta, pamäť haldy, atď. sú zdieľané medzi všetkými vláknami procesu a každé vlánko k nim môže podľa potreby pristupovať. Nie je pochýb, že prístup k spoločným prostriedkom musí byť kontrolovaný a riadený, aby nedochádzalo k chybovosti aplikácie zapríčinenej zmenou premenných, alebo dát. Riadenie prístupu k spoločným prostriedkom má na starosti operačný systém, ktorý rieši prístup vlákien ich synchronizáciou. MS Windows za účelom tohto riešenia poskytuje širokú škálu objektov, kde každý z objektov má svoje výhody a nevýhody s ohľadom na rýchlosť, funkčnosť a schopnosti synchronizácie medzi procesmi [17].

V následujúcich pokapitolách si v krátkosti popíšeme jednotlivé typy synchronizácie vlákien v operačných systémoch Windows a ich súhrnný prehľad spoločne s výhodami a nevýhodami je uvedený v tabuľke prílohy A.2.

5.1 Kritická sekcia

Jedným z najjednoduchších synchronizačných objektov v MS Windows je *kritická sekcia* (*angl.* Critical section), ktorá synchronizuje prístup vlákien k spoločným prostriedkom mechanizmom *exkluzívneho prístupu jedného vlákna*. To znamená, že k zdieľaným dátam môže pristupovať v jednej chvíli len jedno vlákno. Pokiaľ nedochádza počas behu programu ku kolízii pri prístupe do kritickej sekcie, beží kritická sekcia v uživateľskom móde, takže je extrémne rýchla. Za bezkolízneho stavu tento typ synchronizácie nemá takmer žiadnu časovú réžiu, ktorá je bežne spotrebovávaná synchronizáciou vlákien pri prepínaní medzi uživateľským módom a módom jadra. V stave, keď dôjde ku kolízii prístupu viacerých vlákien do kritickej časti, je časová réžia trochu vyššia, nakoľko zodpovednosť a riešenie synchronizácie prístupu padá na mód jadra a behové

prostredie musí predať riadenie z uživateľského módu na mód jadra. Vyššia časová réžia však nie je vo väčšine prípadov nijak významná nakoľko aktuálne vlákno, ktoré má prístup do kritickej sekcie, tak či tak blokuje prístup ostatným vláknam. Blokované vlákna sú v kolíznom stave a musia čakať vo „fronte“ pokiaľ bude kritická časť opäť uvolnená. Jednou z hlavných nevýhod tohto typu synchronizácie je absencia objektu jadra priamo spojeného s kritickou sekciou, následkom čoho je nemožnosť synchronizácie prístupu medzi samostatnými procesmi.

5.2 Mutex

Mutex je ďalším typom synchronizovaného prístupu vlákien k spoločným prostriedkom, ktorý je vo Windows realizovaný ako objekt jadra operačného systému. Tým, že je mutex realizovaný na úrovni módu jadra, rieši hlavný nedostatok a nevýhodu kritickej sekcie, ktorá neumožňuje synchronizáciu medzi viacerými procesmi, ale len synchronizáciu vlákien v rámci jedného procesu. Mutexy teda umožňujú medzi procesovú synchronizáciu, kde je však nutné počítať s väčšou časovou réžiou v prípade kolízneho stavu. Pri každom volaní „wait“ funkcie v kolíznom stave musí totiž mutex prejsť z uživateľského módu do módu jadra, čo je v prípade mutexov časovo náročnejší proces. Rovnako ako kritická sekcia, aj tento typ synchronizácie umožňuje len exkluzívny prístup jedného vlákna, alebo procesu k zdieľaným dáta. Pokiaľ chce vlákno, alebo proces pristupovať k zdieľaným dátam, musí byť vlastníkom mutexu. V prípade, že mutex vlastní v danej chvíli iné vlákno, prejde do „wait“ stavu kde čaká na jeho uvolnenie.

5.3 Semafór

Semafór (*angl.* semaphore) je veľmi podobný mutexu, nakoľko je taktiež implementovaný ako objekt módu jadra. S mutexom rovnako zdieľa, jednak jeho výhodu synchronizácie procesov na úrovni jadra, tak aj s tým spojenú nevýhodu vyššej časovej náročnosti. Hlavným rozdielom medzi mutexom a semafórom je schopnosť väčšej priepustnosti prístupu do kritickej časti u semafóru. Semafór tak neumožňuje len exkluzívny prístup jedného vlákna k zdieľaným prostriedkom, ako kritická sekcia, či mutex. Pri jeho inicializácii môžme parametricky určiť jeho priepustnosť a umožniť tak prístup viacerých vlákien v jednej chvíli k spoločným prostriedkom. Povolenie prístupu viacerých vlákien môže byť samozrejeme nebezpečené a vo väčšine prípadov sa nevyužíva. Existujú však situácie kde je vyššia priepustnosť žiadaná a niekedy až nevyhnutná.

5.4 Udalosť

Udalosti (*angl.* events) sú primitívne synchronizačné objekty na úrovni jadra, na ktorých sú postavené ostatné objekty, respektíve typy synchronizácie vlákien. Sami o sebe sú veľmi pomalé, ale umožňujú synchronizáciu procesov pomocou pomenovaných udalostí.

5.5 Metered sekcia

Metered sekcie (*angl.* metered sections) vychádzajú primárne z kritických sekcií a v podstate sú ich rozšírením. Prvým rozšírením je schopnosť synchronizácie procesov, respektíve synchronizácie vlákien naprieč procesmi, ktoré umožňuje mutex, alebo semafór. Druhým rozšírením je variabilita priepustnosti prístupu k spoločným prostriedkom v kritickej časti, ktorý sme videli u semafóru. Hlavnou myšlienkou pri zrode metered sekcie bolo vytvorenie nového typu synchronizovaného

prístupu, ktorý bude rýchly ako kritická sekcia, bude podporovať medzi procesovú synchronizáciu vlákien, bude umožňovať nastavením regulovateľnú väčšiu priepustnosť k zdieľaným prostriedkom a bude kompatibilná so všetkými OS Windows.

Z uvedených typov synchronizačných prístupov je v dnešnej dobe asi najpokročilejší typ metered section, ktorý v sebe zahŕňa všetky vlastnosti jej predchodcov. Nevýhodou môže byť jeho zložitejšia implementácia vo vlastnom kóde programu. Metered section totiž nie je priamo dostupná v knižniciach Windows, konkrétne *windows.h*, ale je potrebné ju do programu zakomponovať ako samostatnú triedu realizujúcu metered section [18].

6 Dátové štruktúry

Dátové štruktúry sú dnes už dennou rutinou každého programátora a sú obsiahnuté skoro v každej aplikácii. Voľba správnej dátovej štruktúry môže ovplyvniť takmer všetko týkajúce sa aplikácie, počínajúc vplyvom na jej rýchlosť až po prehľadnosť kódu. Dátové štruktúry teda významne rozširujú možnosti programového riešenia, čím sa zvyšuje rozsah prípadných uplatnení informatických metód. Nakoľko sú dátové štruktúry každému známe nebudeme sa s nimi dlho zaoberať a len v krátkosti spomenieme pár základných typov, s ktorými pracuje aj námi vytvorený rámec pre správu pamäte.

Dátových štruktúr je v súčasnosti slušný počet a je možné ich niekoľkými spôsobmi kategorizovať, na základe ich vlastností, štruktúry apod. Nasledujúci prehľad kategorizuje dátové štruktúry do dvoch hlavných skupín, na lineárne a nelineárne [19]:

- *Lineárne dátové štruktúry:*
  - *pole,*
  - *zásobník,*
  - *fronta,*
  - *seznam.*
- *Nelineárne dátové štruktúry:*
  - *strom* (voľný, binárny, B-strom atď.).

6.1 Pole

Najjednoduchšou a najstaršou dátovou štruktúrou je pole (*angl.* array). Jedná sa o súvislí blok vyalokovanej pamäte daného dátového typu, ku ktorého prvkom pristupuje program, počas svojho behu, prostredníctvom indexov. Pole má vo väčšine prípadov pevnú veľkosť, ktorú definujeme pri jeho inicializácii a dynamicky sa mení len jeho obsah. V súčastnosti však poznáme okrem klasického poľa aj pole dynamické, ktorého implementácia umožňuje dynamicky meniť jeho veľkosť počas behu na základe potrieb programu.

6.2 Zásobník

Zásobník je ďalšou dátovou štruktúrou, ktorého implementácia je veľmi často založená na predchádzajúcej štruktúre poľa. Zásobník je priamo prístupný len zo strany tzv. vrcholu, teda prvej voľnej pozície. Z tohto dôvodu sa princíp funkcie zásobníku označuje skratkou *LIFO* (*angl.* Last In First Out), čo vo voľnom preklade znamená „posledný dnu prvý von“. Prvky vkladané do zásobníku sa vždy ukladajú na vrchol a odoberanie prvkov zo zásobníku je vždy opačné, čiže od vrcholu postupne dolu. Zásobník je dátovou štruktúrou s obmedzeným prístupom.

6.3 Fronta

Fronta je dátová štruktúra veľmi podobná zásobníku. Rovnako ako u zásobníku, je prístup k prvkom obmedzený a jeho základom je princíp FIFO (*angl.* First In First Out), čiže prvý dnu prvý von. Prvky sú do fronty vkladané jeden za druhým, čo je taktiež podobné so zásobníkom.

Rozdiel nastáva až pri ich uvolňovaní, kde prvý uvolnený prvok z fronty je ten, ktorý bol do fronty prvý vložený. Princíp je úplne identický s bežnou frontou v hypermarkete.

6.4 Zoznam

Poslednou dátovou štruktúrou, ktorú si predstavíme je zoznam, ktorý má na rozdiel od predchádzajúcich štruktúr väčšiu štrukturálnu aj implementačnú zložitosť a jeho realizácia nevychádza primárne z dátovej štruktúry poľa. Jedná sa teda o pamäťovo úspornú dátovú štruktúru, ktorá dovoľuje zoskupiť ľubovoľný počet prvkov obmedzený len množstvom dostupnej pamäte. Zoznam môže byť jednosmerný, alebo obojsmerný a jeho základom sú uzly obsahujúce mimo dátovú položku i ukazovatele na predchádzajúci a následujúci uzol. Zoznam tak tvorí „reťazec“ navzájom pospájaných uzlov. Jeho implementácia môže byť rôzna v závislosti od potrieb programu, ale v skutku veľmi pestrá. Štruktúra a princíp zoznamu totiž umožňujú implementáciu množstva pomocných a užitočných funkcií, ako je triedenie zoznamu, prístup k rôznym prvkom zoznamu, odstránenie bloku prvkov zo zoznamu, presun bloku prvkov z jedného miesta na druhé, vloženie prvku na konkrétnu pozíciu atď [20].

7 Analýza a návrh

Praktickou časťou diplomovej práce bolo vytvoriť rámec pre vytváranie dátových štruktúr v hlavnej pamäti pomocou C++. V tejto kapitole sa budeme zaoberať návrhom implementácie rámca. Návrh rámca je rozdelený do dvoch, respektíve do troch častí. Prvá časť rieši návrh správcu pamäte, ktorého úlohou bude alokácia a správa pamäťových blokov, ich prideľovanie a uvolňovanie jednotlivým žiadateľom a riešenie konkurenčného prístupu. Druhá časť sa venuje analýze a návrhu dátových štruktúr, ktoré budú pre svoje potreby využívať výhradne pamäť pridelenú správcom pamäte rámca. V tretej časti si popíšeme návrh testov, pomocou ktorých otestujeme celé riešenie vytvoreného rámca. Výsledkom a záverom z testovania sa budeme podrobnejšie venovať v poslednej 9.kapitole.

Pred tým, ako budeme pokračovať samotnou analýzou a návrhom rámca, upresníme si naše zadanie, nakoľko námi vytváraný rámec sa bude do určitej miery odlišovať od bežných pamäťových správcov implementovaných na aplikačnej úrovni. Hlavnou úlohou rámca je automatická správa pamäte u jednoduchých dátových štruktúr pomocou C++. Z toho vyplýva, že našou úlohou nie je navrhnúť a implementovať riešenie, ktoré bude plne nahradzovať bežnú manipuláciu s pamäťou na halde pomocou `new` a `delete`, ale poskytnúť užívateľovi alternatívu k predvoleným možnostiam alokácie a uvolňovania pamäte na halde. Pri práci, tak bude mať programátor možnosť alokovať a uvolňovať pamäť pomocou rozhrania námi implementovaného rámca, ktorý bude pracovať s pamäťovými blokmi na aplikačnej úrovni. Výsledkom toho by mal byť efektívnejší prístup k pamäti a s tým spojené celkové zrýchlenie aplikácie. Okrem jednoduchého prístupu k pamäťovým blokom bude mať programátor k dispozícii základné dátové štruktúry implementované rámcom, ktoré budú výhradne pracovať s pamäťovými blokmi spravovanými samotným rámcom. Dátové štruktúry rámca, by tak mali pracovať s pamäťou efektívnejšie v porovnaní s bežnými dátovými štruktúrami využívajúcimi predvoleného správcu pamäte v C++.

7.1 Návrh správcu pamäte

V tejto kapitole sa budeme venovať analýze a návrhu časti rámca, tzv. `MemoryManageru`, ktorého úlohou bude alokácia a správa pamäťových blokov, ich prideľovanie, uvolňovanie a riešenie konkurenčného prístupu u viacvláknových aplikácií. Rozhranie `MemoryManagera` bude poskytovať mimo implicitnej inicializácie, aj základné možnosti nastavenia jeho vlastností prostredníctvom parametrov konštruktoru. Samozrejmosťou bude schopnosť rozšírenia jeho pamäťových prostriedkov o ďalšie voľné pamäťové bloky, v závislosti na jeho nastavení.

Pri návrhu správcu pamäte sa nám naskytuje niekoľko základných vlastností, či charakteristík, ktorými môže správca pamäte disponovať a ktoré bude následne poskytovať ako služby ostatným aplikáciám, ktoré s ním budú pracovať. Správca pamäte na aplikačnej úrovni má dve hlavné dimenzie možných riešení, v ktorých musíme učiniť rozhodnutie, akým smerom sa bude námi vytváraný správca pamäte uberať a aké služby bude ponúkať. Jedná sa o riešenie veľkosti pamäťových blokov a riešenie súbežného prístupu.

V prípade veľkosti pamäťových blokov sa naskytujú dve možnosti prístupu k ich realizácii, kde každá možnosť má určité výhody a nevýhody:

- *Fixná veľkosť pamäťových blokov* – Pamäťové bloky spravované správcom budú mať fixnú veľkosť. Výhodou tohto riešenia je vyšší výkon správcu pamäte nakoľko sa

odbúra nadmerné prepočítavanie pri požiadavkách rôznej veľkosti a hierarchizácia pamäťových blokov. Nevýhodou je vznik internej fragmentácie pri nevhodnom použití správcu pamäte a v našom prípade i celého rámca.

- *Premenlivá veľkosť pamäťových blokov* – Pri tomto riešení bude správca pamäte schopný alokovať a prideľovať pamäťové bloky rôznych veľkostí. Výhodou riešenia je do istej miery efektívnejšie nakladanie s pamäťou na halde a možnosť realizácie správcu pamäte, ktorý bude transparentnejší pre programátora. Nevýhodou je vyššia časová i priestorová réžia potrebná pre pokročilejšie algoritmy a spomalenie uvoľňovania pamäte pri úplnom návrate pamäťových blokov na haldu. Toto riešenie je vhodnejšie pre správcu pamäte, ktorý by mal plne nahradzovať štandardného správcu pamäte behového prostredia.

Naše rozhodnutie v tejto chvíli do istej miery ovplyvňuje špecifickejšie zameranie celého rámca, ktorého primárnou úlohou nie je nahradiť správu pamäte behového prostredia C++, ale realizovať vytváranie dátových štruktúr v hlavnej pamäti. Z tohto dôvodu budeme správcu pamäte realizovať s fixnou veľkosťou pamäťových blokov na troch úrovniach. To znamená, že správca pamäte bude pracovať s tromi typmi pamäťových blokov: SMALL, BIG, SYSTEM, kde každý typ bude mať svoju vlastnú veľkosť. Veľkosť jednotlivých pamäťových blokov bude pri inicializácii `MemoryManageru` nastavená, buď implicitne na hodnoty pevne dané rámcom, alebo na užívateľom zadané hodnoty pomocou parametrov konštruktoru.

Riešenie pokrývajúce premenlivú veľkosť pamäťových blokov by mohlo byť námetom pre ďalšie rozšírenie rámca, kde by bolo vhodné vytvoriť druhé riešenie ako samostatný modul, ktorý by svojou väčšou časovou a priestorovou réžiou nedegradoval výkon pamäťového správcu s fixnou veľkosťou pamäťových blokov.

Rovnako, ako u veľkosti pamäťových blokov, aj u súbežného prístupu existujú dva pohľady na problém a jeho riešenie. Správca pamäte môže mať súbežný prístup riešený:

- *S podporou jednovláknových procesov* – tento typ správcu pamäte bude obmedzený len na jednovláknové procesy, bez akejkoľvek podpory a použitia s viacvláknovými aplikáciami.
- *S podporou viacvláknových procesov* – správca pamäte bude riešiť súbežný prístup viacerých vlákien k správe pamäte pomocou synchronizačných objektov dostupných vo Windows API.

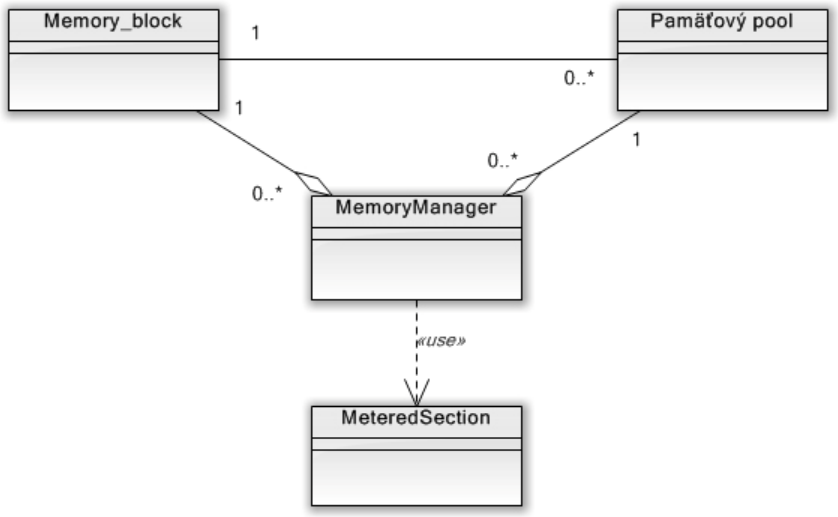
Pri výbere riešenia súbežného prístupu je voľba jednoznačná, nakoľko je našou úlohou vytvoriť rámec, ktorý bude prideľovať bloky pamäte jednému, alebo viacerým externým procesom súčasne. A čo viac, podpora len jednovláknových procesov by bola veľkým obmedzením v použití rámca.

Po vstupných rozhodnutiach sa nám podarilo načrtnúť základ správcu pamäte rámca, ktorý bude mať nasledujúce vlastnosti. Správca bude schopný alokovať z hlavnej pamäte, spravovať a prideľovať pamäťové bloky dvoch, respektíve troch veľkostí, ktoré budú závislé na jeho nastavení a bude spoľahlivo riešiť konkurenčný prístup viacerých vlákien, ktoré budú súčasne žiadať o pamäť.
K tomu, aby bol navrhovaný správca schopný prevádzať spomínané činnosti bude potrebovať niekoľko podporných objektov a niekoľko pamäťových poolov. Základný statický pohľad na

abstraktné objekty správcu pamäte je vidieť na obrázku 5, kde `MemoryManager` reprezentuje centrálny objekt správcu pamäte, `MeteredSection` objekt pre synchronizáciu viacvláknového prístupu a `Memory_block` jednotlivé pamäťové bloky, ktoré budú vkladané do pamäťového poolu. Aby sme dosiahli maximálny výkon správy pamäte rámca, je dôležité aby alokácia všetkých objektov prebiehala v pamäti spravovanej sebou samým. Jednoducho povedané, jeho réžia by mala byť pokiaľ možno s čo najmenším počtom ďalších alokácií a dealokácii na halde prostredníctvom príkazov `new` a `delete`. K halde by mal správca pamäte pristupovať len počas jeho inicializácie a pri rozšírení pamäťových poolov, v prípade nedostatku voľných pamäťových blokov.

Vnútorná logika správcu pamäte bude pozostávať zo štyroch základných procesov:

- *inicializácia správcu pamäte* a alokácia pamäťových poolov,
- *prideľovanie pamäťových blokov* externým procesom,
- *návrat uvolnených pamäťových blokov,*
- *rozšírenie pamäťových poolov* pri nedostatku voľných pamäťových blokov.



Obrázok 5: Statický pohľad na navrhovaného správcu pamäte rámca.

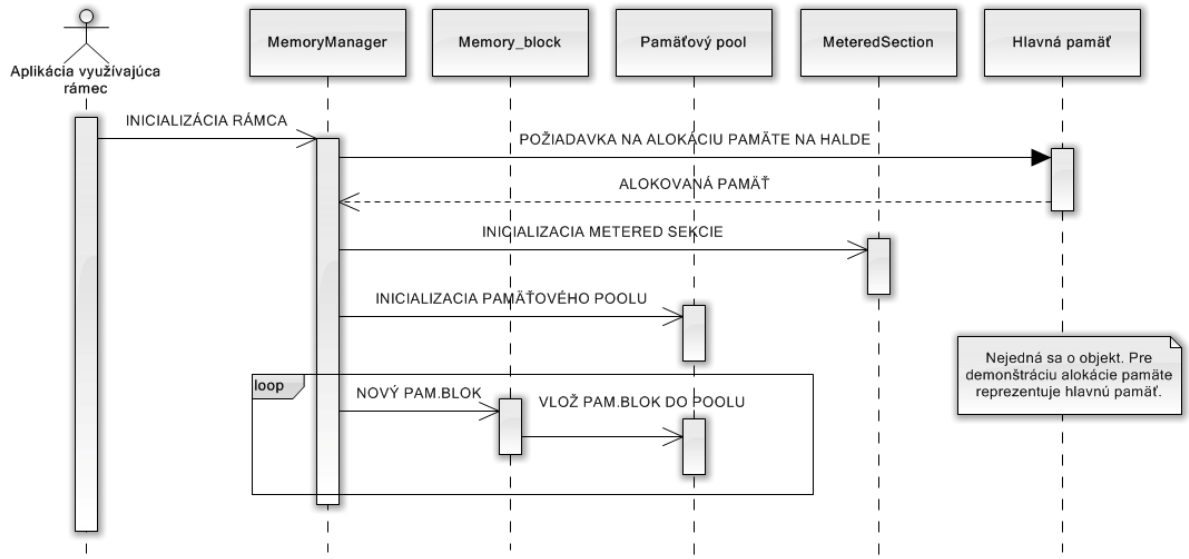
Proces inicializácie správcu pamäte a alokácia pamäťových poolov

Základným procesom správcu pamäte je jeho inicializácia, ktorá bude dostupná prostredníctvom sady konštruktorov umožňujúcich variabilné nastavenie štruktúry a chovania správcu pamäte. Vstupnými parametrami jednotlivých konštruktorov si bude môcť programátor nadefinovať množstvo na počiatku alokovaných pamäťových blokov, veľkosti blokov v jednotlivých skupinách a určiť *režim správcu pamäte*. Správca pamäte bude z pohľadu rozšírenia pamäťových poolov fungovať v dvoch režimoch:

- *režim povoleného rozšírenia pamäťových poolov,*
- *režim zamietnutého rozšírenia pamäťových poolov.*

V prípade nedostatku pamäťových blokov, bude správca pamäte v prvom režime alokovať ďalšiu pamäť na halde a rozširovať konkrétny pamäťový pool o ďalšie bloky. V druhom režime zamietnutého rozšírenia, nebude správca pamäte alokovať ďalšiu pamäť na halde a externému procesu žiadajúcemu o blok pamäti vráti prázdnu hodnotu.

Súčasťou inicializácie bude vyalokovanie potrebného množstva pamäte na halde a jej namapovanie do jednotlivých pamäťových blokov, kde každý blok pamäte bude realizovaný vlastným objektom `Memory_block` a bude poskytovať základné údaje o bloku pamäte. Vlastnosti pamäťového bloku musia byť v takom rozsahu, aby boli nápomocné pre ich efektívnu správu, ale aj pre jednoduchšiu manipuláciu s pamäťovým blokom v externých procesoch. Jedná sa o informácie o veľkosti bloku, jeho aktuálnom stave, začiatku a konci bloku z pohľadu adresného priestoru, využití bloku a pod. Ďalším krokom bude organizácia vytvorených pamäťových blokov do jednotlivých pamäťových poolov, tvorených frontami založenými na jednosmernom zozname. Každý pamäťový blok sa tak po svojej inicializácii zaradí do príslušného pamäťového poolu, kde bude čakať na jeho pridelenie externým procesom (obrázok 6). Toto programové riešenie pamäťových poolov bude umožňovať manipuláciu s pamäťovými blokmi (odoberanie a návrat) vo vnútri každého poolu s časovou zložitosťou *O(1).*



Obrázok 6: Sekvenčný diagram inicializácie správcu pamäte.

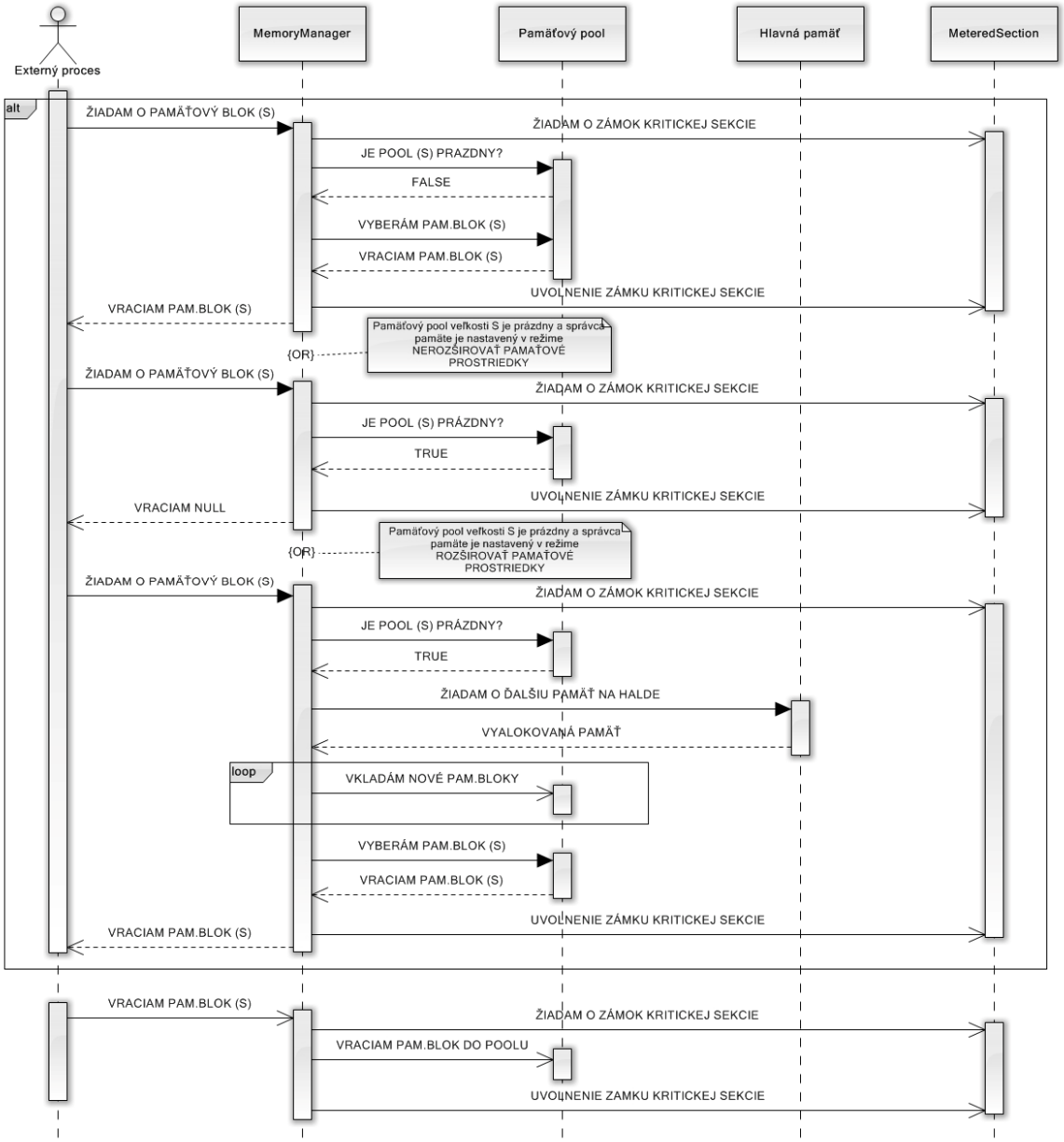
Prideľovanie pamäťových blokov externým procesom

Externé procesy budú môcť požiadať o blok pamäte prostredníctvom kolekcie preťažených metód, ktoré budú umožňovať explicitný výber veľkosti požadovaného bloku. Každý blok pamäťového poolu bude objektom triedy `Memory_block`. Pri pridelení pamäte na základe požiadavky externého procesu sa z príslušného pamäťového poolu odoberie blok, označí sa za obsadený a odkaz na tento blok bude predaný žiadateľovi. Pre správcu pamäte sa stane takto pridelený blok už nedostupný v rámci jeho metód prideľovania pamäťových blokov. Zostáva samozrejme stále prítomný v pamäti alokovanej správcom počas jeho inicializácie, takže jeho „majiteľom“ zostáva naďalej správca pamäte, blok už však nebude dostupný v príslušnom pamäťovom poole. Späť do

pamäťového poolu sa dostane až po jeho navrátení externým procesom, ktorý je jeho dočasným vlastníkom. V prípade, ak dôjde k situácii, že pamäťový pool danej veľkosti bude prázdny, v závislosti na nastavení správcu pamäte sa spustí proces rozšírenia (obrázok 7).

Návrat uvolněných pamäťových blokov

Externé procesy budú mať samozrejeme možnosť blok pamäte, ktorý majú vo svojom vlastníctve, vrátiť správcovi pamäte. Pri návrate pamäťového bloku sa prevedie opačný postup, ako v procese prideľovania. Blok sa označí za prázdny a zaradí sa späť do pamäťového poolu (obrázok 7).



Obrázok 7: Sekvenčný diagram zobrazujúci alternatívy pridelenia a uvolnenia pamäťového bloku veľkosti S.

Nakoľko bude námi vytváraný rámec podporovať viacvláknové procesy, musia mať procesy prideľovania a uvolňovania pamäťových blokov spoľahlivo vyriešený konkurenčný prístup. U viacvláknových aplikácii by totiž mohlo dochádzať ku konfliktom pri súbežnom prístupe dvoch, alebo viacerých vlákien k pamäťovým poolom rámca. Procesom dvoch, alebo viacerých vlákien by mohol byť v čase konfliktu pridelený rovnaký blok pamäte a vlákna by si navzájom prepisovali dáta. Nevýhodou je, že k tejto chybe by mohlo dôjsť až po určitej dobe behu programu a chyba by nemusela byť na prvý pohľad zrejmá.

Konkurenčný prístup budeme riešiť objektom `MeteredSection` pre vzájomnú synchronizáciu vlákien v OS Windows, ktorý sme si ukázali v 5. kapitole. Pred každým volaním funkcie pre pridelenie, alebo uvolnenie pamäťového bloku, bude predchádzať požiadavka o zámok a vstup do kritickej sekcie. Po prevedení operácie s pamäťovým blokom bude samozrejme zámok explicitne uvolnený. Metered sekcia správcu pamäte bude implicitne nastavená inicializačnými parametrami na režime exkluzívneho prístupu a počas behu programu nebude možné režim meniť.

Rozšírenie pamäťových poolov pri nedostatku voľných pamäťových blokov

V prípade, že nastane situácia kedy je pri požiadavke o blok pamäte danej veľkosti pamäťový pool prázdny, dôjde k alokácii ďalšej pamäte na halde a k rozšíreniu pamäťového poolu. Spustenie procesu rozšírenia bude však závislé na nastavení správcu pamäte, kedy bude môcť programátor určiť či chce používať rámec v režime povoleného rozšírenia, alebo nie. Možnosť režimu rozšírenia sme zaviedli do architektúry správcu pamäte aj z dôvodu otestovania korektnosti rámca, ku ktorému sa bližšie dostaneme v kapitole 9.

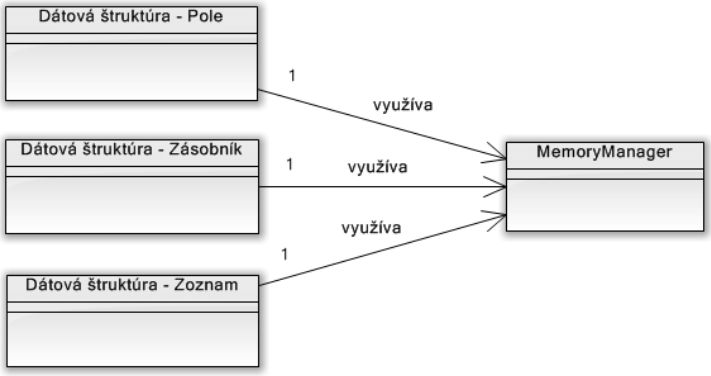
7.2 Návrh dátových štruktúr rámca

Ďalšou časťou návrhu rámca pre vytváranie dátových štruktúr v hlavnej pamäti pomocou C++ je návrh dátových štruktúr, ktoré budú schopné sa efektívne a rýchlo alokovať z hlavnej pamäte. Navrhované štruktúry budú k rýchlej alokácii a dealokácii využívať služby námi navrhnutého správcu pamäte, u ktorého budú žiadať o pamäťové bloky. Dátové štruktúry rámca budú disponovať rozhraním poskytujúcim ich základnú funkčnosť, ktoré bude zodpovedať ich bežnej implementácii a bude umožňovať štandardnú prácu s dátovou štruktúrou. Tým máme na mysli bežné metódy, ako napríklad pridanie prvkov, prístup k jednotlivým prvkom, odobranie prvkov, rušenie štruktúry a pod., ktoré budú samozrejme špecifické pre každý typ dátovej štruktúry.

Výsledný rámec bude realizovať a poskytovať následujúce dátové štruktúry: pole, zásobník a zoznam (obrázok 8). Frontu bude možné realizovať pomocou zoznamu, ktorý bude disponovať metódou rozhrania pre odobranie prvého prvku zo zoznamu s návratovou hodnotou. Každá dátová štruktúra bude tvorená samostatnou triedou, ktorá bude minimálne realizovať následujúce procesy:

- *inicializácia dátovej štruktúry,*
- *vloženie prvku do dátovej štruktúry,*
- *prístup k jednotlivým prvkom dátovej štruktúry,*
- *odobranie prvku z dátovej štruktúry,*
- *rozšírenie pamäťových prostriedkov,*
- *nulovanie obsahu dátovej štruktúry,*
- *kompletné uvoľnenie dátovej štruktúry.*

Hore uvedený zoznam uvádza základné procesy, ktorými bude disponovať každá z dostupných dátových štruktúr. Výpis všetkých procesov u každého dátového typu je uvedený v tabuľke prílohy A.3.



Obrázok 8: Statický pohľad na vzťah dátových štruktúr rámca so správcom pamäte.

Inicializácia dátovej štruktúry

Jednotlivé dátový štruktúry bude možné vytvoriť, buď priamo na halde, alebo v pamäťovom bloku rámca. Inicializácia dátových štruktúr bude realizovaná kolekciou konštruktorov, ktoré budú poskytovať variabilitu počiatočného nastavenia dátovej štruktúry. Nastavením dátovej štruktúry je myslené určenie veľkostného typu blokov pamäte, s ktorými bude dátová štruktúra pracovať. Programátor si tak bude môcť určiť, podľa potreby a danej situácie, s akými pamäťovými blokmi bude ním inicializovaná dátová štruktúra pracovať. Objekty jednotlivých dátových štruktúr bude možné alternatívne inicializovať prostredníctvom prázdneho konštruktoru a jednou z kolekcie preťažených inicializačných metód, ktoré budú nahradzovať jednotlivé konštruktory. Alternatívna inicializácia dátových štruktúr je navrhnutá za účelom vytvorenia objektu priamo v pamäťovom bloku pridelenom správcom pamäte, použitím pretypovania, čím sa opäť redukuje priama alokácia na halde.

Počas inicializácie dátových štruktúr si každý objekt vytvorí vlastný, *súkromný pamäťový pool*, ktorý sa bude plniť objektu pridelenými pamäťovými blokmi. Pamäťový pool má samozrejme taktiež určitú pamäťovú réžiu potrebnú k jeho administrácii. Pre tieto administratívne účely si dátová štruktúra požiadať o ďalšie pamäťové bloky, ešte pred inicializáciou súkromného pamäťového poolu. Pamäťový pool objektu musí obsahovať minimálne jeden pamäťový blok príslušnej veľkosti (obrázok 9).

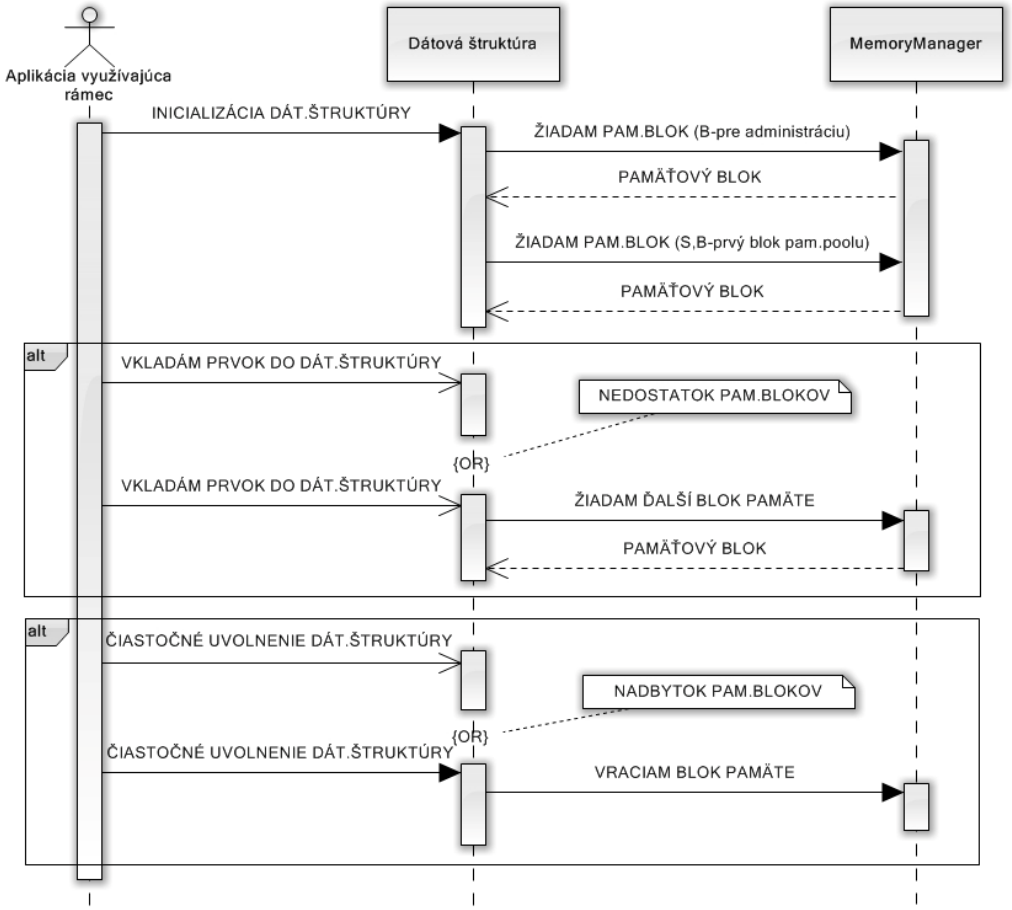
Vloženie prvku do dátovej štruktúry

Vloženie prvku u jednotlivých dátových štruktúr bude realizované metódami, ktorých názvy budú identické s názvami z bežných implementácii dátových štruktúr. Detailná realizácia spolu s názvom metódy bude závisieť od konkrétnej dátovej štruktúry. U zásobníku to bude napríklad metóda `push`, ktorá bude vkladať prvok na vrchol zásobníku. Vložené prvky budú postupne ukladané do pamäťových blokov nachádzajúcich sa v súkromnom pamäťovom poole. V prípade nedostatočnej kapacity súkromného pamäťového poolu, bude objekt schopný jeho realokácie, pomocou procesu realokácie (obrázok 9).

Prístup k jednotlivým prvkom dátovej štruktúry
Prístup k prvkom dátovej štruktúry bude realizovaný individuálne podľa typu a charakteristiky konkrétnej dátovej štruktúry, rovnako ako je prístup k prvkom realizovaný v jej bežnej implementácii. U zásobníku posledný vložený prvok bude prvý vybraný, u poľa budú prvky dostupné pomocou operátoru [] apod.

Odobranie prvku z dátovej štruktúry – Rovnako, ako v predchádzajúcom procese, bude aj postup odobrania prvku závislý na konkrétnej dátovej štruktúre. Pri masívnom odobraní väčšieho množstva prvkov, prípadne u hromadného odobrania prvkov v zozname, bude objekt dátovej štruktúry schopný uvolniť nevyužívané a teda nadbytočné pamäťové bloky a vrátiť ich späť správcovi pamäte.

Rozšírenie pamäťových prostriedkov
V prípade nedostatočnej kapacity súkromného pamäťového poolu bude objekt schopný požiadať správcu pamäte o ďalší voľný blok. Pridelený blok sa uloží a zaradí do súkromného pamäťového poolu objektu a objekt s ním bude môcť pracovať pri svojej ďalšej činnosti.



Obrázok 9: Inicializácia, pridanie a odobranie prvkov z dátovej štruktúry poľa `MF_Array` znázornené pomocou sekvenčného diagramu.

Nulovanie obsahu dátovej štruktúry
Proces nulovania dátovej štruktúry bude možné realizovať dvomi spôsobmi. V prvom prípade bude proces vracať objekt dátovej štruktúry do jeho počiatočného stavu, kedy budú správcovi pamäte vrátené všetky pamäťové bloky okrem prvého bloku a ukazovatele sa vrátia do počiatočného stavu. „Nulovaná“ dátová štruktúra sa jednoducho vyprázdni. V druhom prípade, proces ponechá parametrom zadaný počet prvkov v dátovej štruktúre a ostatné pamäťové bloky vráti správcovi.

Kompletné uvolnenie dátovej štruktúry
Jedná sa v podstate o finalizáciu objektu, kedy objekt vráti správcovi pamäte spoločne so všetkými pamäťovými blokmi zo súkromného poolu aj bloky, ktoré využíval k svojej administrácii.

7.3 Návrh testovacích prípadov

K testom sa podrobnejšie dostaneme v poslednej kapitole, kde si popíšeme ich skutočnú realizáciu spoločne s výsledkami. V tejto kapitole si len nadefinujeme základné členenie konečných testov s krátkym popisom ich účelu.

Testovacie prípady u nášho rámca a ich samotná realizácia bude trochu odlišná od bežného testovania užívateľských aplikácii, nakoľko s rámcom nebude pracovať priamo koncový užívateľ. Pri kontrole riešenia a správnej funčnosti rámca budeme realizovať dva typy testov:

- *výkonnostné testy* (*angl.* benchmarking),
- *testy korektnosti* (*angl.* correctness testing).

Testy jednotlivých skupín by nám mali pomôcť overiť správnosť a spoľahlivosť konečného riešenia rámca a ukázať jeho výhody v porovnaní s bežným prístupom k pamäti dostupnom v behovom prostredí C++. U výkonových testov budeme primárne testovať rámec z pohľadu dátových štruktúr rámca a v teste korektosti overíme spoľahlivosť správcu pamäte pri manipulácii s pamäťovými blokmi počas viacvláknového prístupu.

8 Implementácia

V ôsmej kapitole sa budeme venovať samotnej implementácii rámca pre vytváranie dátových štruktúr v hlavnej pamäti. V prvej podkapitole si spomenieme technológie, ktoré boli použité pri implementácii, v krátkosti sa teda pobavíme o programovacom jazyku C++ a objekte metered section pre synchronizáciu vlákien v OS Windows. V druhej podkapitole sa bližšie pozrieme na samotnú implementáciu rámca vychádzajúcu z jeho návrhu.

8.1 Použité technológie

Ako vyplýva už zo samotného názvu, práca bola realizovaná v programovacom jazyku C++. K implementácii sme využili vývojové prostredie Microsoft Visual Studio 2010 Professional, ktoré je v súčastnosti najsilnejším vývojovým prostredím platformy Windows, s rozsiahlou užívateľskou základňou a podporou procesu vývoja softwarových aplikácií od návrhu ich architektúry, cez tvorbu, testovanie až po ich nasadenie. Pre konkurenčný prístup, respektíve synchronizáciu vlákien bol implementovaný objekt metered section.

8.1.1 Programovací jazyk C++

C++ je objektovo orientovaný jazyk vyvinutý Bjarnom Stroustrupom a ďalšími v Bellových laboratóriách AT&T. C++ podporuje niekoľko programovacích paradigmat, ako je procedurálne programovanie, objektovo orientované programovanie a generické programovanie. Nie je teda jazykom čisto objektovým.

C++ vznikol rozšírením jazyka C, ku ktorého základni pripája mnohé rozšírenia v podobe tried, virtuálnych funkcií, preťažovania operátorov, viacnásobnej dedičnosti, šablón, spracovania výnimiek apod. Historicky bola norma jazyka C++ po rokoch jeho vývoja prvý krát ratifikovaná v roku 1998, ako ISO/IEC 14882:1998, prvá zmena normy prišla v roku 2003 na normu ISO/IEC 14882:2003 a posledným rozšírením, neformálne známym ako C++11, je norma ISO/IEC 14882:2011 zo septembra minulého roku 2011 [21].

C++ je teda skutočne silným, rozšíreným a obľubeným jazykom, ktorý našiel a naďalej nachádza svoje uplatnenie v rôznych odvetviach a smeroch IT.

8.1.2 Metered sekcie

Metered sekcie (*angl.* metered sections) je objekt určený k efektívnej synchronizácii dvoch, alebo viacerých vlákien pristupujúcich do kritickej oblasti programu. Metered sekcie boli vyvinuté už v roku 1998 pánom Dan Chouom a v súčastnosti sú posledným stupňom vývojovej hierarchie synchronizačných objektov v OS Windows. Bližšie sme sa metered sekciám venovali v 5. kapitole teoretickej časti.

8.2 Vlastná implementácia

Keďže vlastná implementácia vychádza z návrhu rámca, je rovnako rozdelená do troch základných častí, respektíve modulov. Prvý modul je tvorený správcom pamäte, druhý modul tvoria dátové štruktúry rámca a posledný modul tvoria triedy testovacích prípadov. Prvý modul správcu pamäte je schopný aj samostatnej existencie, čiže programátor využívajúci rámec tak môže pracovať so správcom priamo, bez potreby ďalších komponent. Ďalšie dva moduly, modul dátových štruktúr

a modul testovacích prípadov sú naopak, plne závislé na správcovi pamäte a ich samostatné použitie nie je možné. Odkaz na správcu pamäte je dokonca jedným z inicializačných parametrov konštruktorov dátových štruktúr rámca, bez ktorých nie je možné jednotlivé objekty incializovať.

8.2.1 Implementácia modulu správcu pamäte

Modul pamäťového správcu je tvorený štyrmi triedami: `MemoryManager, Memory_block, LinkedList a MeteredSection`, ktoré vychádzajú z návrhu v kapitole 7.1. Centrálnym prvkom je trieda `MemoryManager`, ktorá spravuje pamäť alokovanú z haldy, všetky pamäťové pooly a konkurenčný prístup vlákien. Trieda `MemoryManager` obsahuje okrem súkromných metód určených pre administráciu správcu pamäte aj kolekciu verejných metód pre prístup externých procesov k pamäťovým blokom. Preťažená metóda `Memory_request` umožňuje externým procesom vzniesť požiadavku o pridelenie pamäťového bloku, ktorého veľkosť môže byť implicitná, alebo zvolená parametrom metódy. K uvoľňovaniu pamäťových blokov využívajú externé procesy metódu `Release_memory`, ktorá zaradí vrátený pamäťový blok späť do príslušného pamäťového poolu.

Počas inicializácie si `MemoryManager` vyalokuje potrebnú pamäť z hlavnej pamäte na základe veľkostí blokov `SMALL`, `BIG`, `SYSTEM` a každému pamäťovému bloku vytvorí objekt triedy `Memory_block`. Objekt typu `Memory_block` nesie informácie o pamäťovom bloku, ako je veľkosť bloku, príznak bloku, stav bloku, jeho počiatočnú a koncovú adresu v pamäti a ukazovateľ využitia bloku.

Metódy `Memory_request` a `Release_memory` sú „úzkym hrdlom“ správcu pamäte, nakoľko pristupujú priamo k pamäťovým poolom správcu. Obidve metódy v sebe implementujú vstup do kritickej sekcie `MeteredSection`, podľa návrhu z kapitoly 7.1.

8.2.2 Implementácia modulu dátových štruktúr rámca

Modul dátových štruktúr pozostáva výhradne z tried reprezentujúcich jednotlivé dátové štruktúry. V priebehu implemenácie sme sa snažili dodržať názvoslovie rozhrania jednotlivých typov dátových štruktúr, tak ako sú všeobecne známe z bežných implementácii. Triedy dátových štruktúr sú postavené a implementované na typovo zabezpečených parametrizovaných typoch, respektíve šablónach, čo umožňuje ich flexibilitu využitia. Žiadna z dátových štruktúr rámca tak nie je viazaná len na konkrétny dátový typ. Štruktúry sú schopné pracovať s akýmkoľvek objektom a dátovým typom, aj uživateľsky definovaným.

Počas implementácie algoritmu spravujúceho jednotlivé *subelementy* pamäťových blokov u lineárnych dátových štruktúr rámca, sa nám naskytli dve možnosti ich riešenia. Za účelom zistenie, ktorý prístup bude efektívnejší sme sa rozhodli do rámca zapracovať obidva spôsoby ich realizácie. Modul tak obsahuje v prípade dátových štruktúr typu pole a zásobník dve rozdielne triedy, kde každá trieda realizuje inú myšlienku práce so subelementami pamäťových blokov. V konečnom dôsledku sú obsahom modulu dátových štruktúr rámca následujúce triedy:

- `C_Array` – trieda realizujúca bežnú implementáciu poľa pomocou funkcii `malloc()` a `memcpy()`. Trieda bola implementovaná za účelom otestovania riešenia pomocou výkonnostných testov.
- `MF_Array` – trieda realizujúca dátovú štruktúru rámca typu pole.

- `MF_Array2` – trieda realizujúca dátovú štruktúru rámca typu pole s alternatívnym riešením prístupu k subelementom pamäťových blokov.
- `MF_Stack` – trieda realizujúca dátovú štruktúru rámca typu zásobník.
- `MF_Stack2` – trieda realizujúca dátovú štruktúru rámca typu zásobník s alternatívnym riešením prístupu k subelementom pamäťových blokov.
- `MF_List` – trieda realizujúca dátovú štruktúru rámca typu zoznam.

Metódy definované v návrhu rámca, ktorých prehľad sa nachádza v tabuľke prílohy A.3, boli v plnom rozsahu implementované v príslušných triedach modulu dátových štruktúr rámca.

Obsahom triedy `C_Array` je riešenie dynamického poľa pomocou funkcii známych z jazyka C: `malloc()` a `memcpy()`. Trieda bola implementovaná za účelom otestovania riešenia, kde počas výkonnostných testov budeme porovnávať rýchlosť alokácie a dealokácie dátových štruktúr rámca v porovnaní s dátovou štruktúrou `C_Array` a bežnou alokáciou poľa pomocou `new` a `delete`.

Triedy `MF_Array`, `MF_Stack` a `MF_List` pracujú výhradne s pamäťovými blokmi pridelenými správcom pamäte, čím splňujú ďalšiu požiadavku z návrhu, aby dátové štruktúry rámca pracovali s pamäťou čo najefektívnejšie. Počas inicializácie a činnosti objektov týchto tried, tak nedochádza k žiadnym ďalším alokáciám na halde pomocou `new/delete`. Dátové štruktúry ukladajú a pristupujú k svojim prvkom prostredníctvom ukazovateľa v pamäťovom bloku súkromného poolu, ktorého poloha sa pri sekvenčnom vkladaní prvkov do dátovej štruktúry posúva doprava v rámci pamäťového bloku. Poloha ukazovateľa v rámci súkromného pamäťového poolu je daná dvojicou údajov:

- *identifikátorom aktívneho bloku* pamäťového poolu, ktorým ukazuje na posledný pridaný blok do súkromného pamäťového poolu,
- z *pozície ukazovateľa* v rámci aktívneho pamäťového bloku.

U ostatných operácii, kedy sa pristupuje a pracuje s prvkami uprostred dátovej štruktúry je pozícia prvku vždy vypočítavaná na základe dvoch hore uvedených údajov a veľkosti parametrizovaného dátového typu inicializovanej dátovej štruktúry. U týchto operácii sa v podstate jedná o pokročilejšiu aritmetiku ukazovateľov. Nevýhodou tohto riešenia je väčšia réžia pri prístupe k prvkom poľa, kde musí algoritmus takmer pri každom prístupe vypočítavať pozíciu prvku v jednotlivých pamäťových blokoch súkromného poolu. Na druhú stranu, výhodou riešenia je rýchlejšia rozšírenie dátovej štruktúry pri vkladaní väčšieho množstva prvkov.

Triedy `MF_Array2` a `MF_Stack2` majú trochu odlišný mechanizmus prístupu k subelementom pamäťových blokov a pri alokácii a každom rozšírení si vytvárajú nové, súvislé pole ukazovateľov. Pole ukazovateľov je alokované na halde a jeho veľkosť musí byť dostatočne veľká, aby ukazovatele, daného dátového typu inicializovanej dátovej štruktúry, pokryli všetky subelementy blokov súkromného pamäťového poolu. Po alokácii poľa sa jednotlivé ukazovatele inicializujú, ako odkazy do pamäte blokov súkromného poolu. U všetkých operácií, tak dátová štruktúra pristupuje k svojim prvkom, respektíve k pamäti súkromného pamäťového poolu, pomocou jediného, súvislého poľa ukazovateľov. Prístup k prvkom, tak nie je nutné pri každom prístupe zložito vypočítavať, ako tomu bolo v prechádzajúcom prípade. Výhodou riešenia je teda rýchlejší prístup k prvkom dátovej štruktúry. Nevýhodou riešenia je komplikovanejšie rozširovanie súkromného pamäťového poolu o ďalší blok pamäte, kedy sa musí vždy realokovať i pole

ukazovateľov, čo bude mať určite záporný vplyv na rýchlosť dátovej štruktúry u sekvenčného vkladania väčšieho množstva prvkov. Rozdiel v prístupe k prvkom u jednotlivých skupín dátových štruktúr je vidieť na obrázku 10.

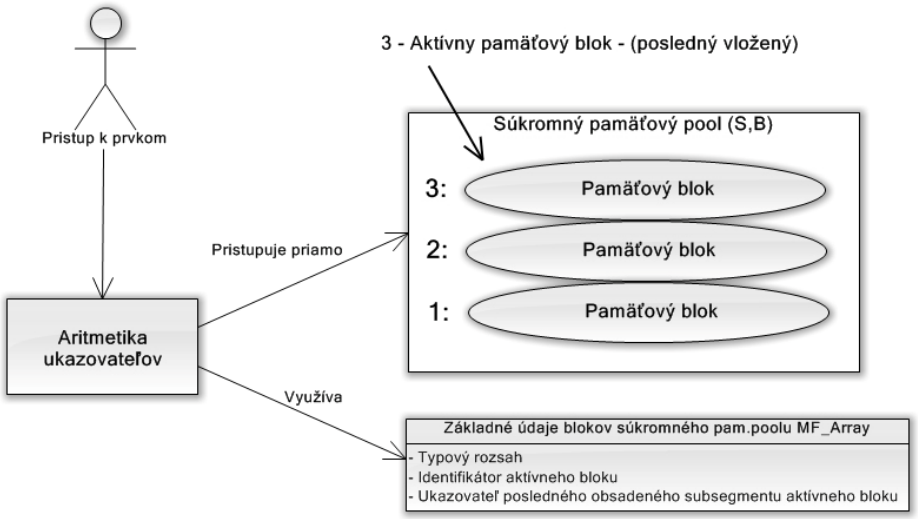
Každé z riešení má svoje výhody a nevýhody o ktorých sa presvedčíme v závere práce u výkonnostných testov. V nasledujúcich odstavcoch sa budeme snažiť popísať hlavný rozdiel medzi `MF_Array/MF_Array2` a `MF_Stack/MF_Stack2`.

Mechanizmus a algoritmy sú si v obidvoch prípadoch, u dátovej štruktúry poľa i zásobníku, veľmi podobné. Z tohto dôvodu si popíšeme rozdiely len u dátovej štruktúry typu pole, reprezentovaného triedami `MF_Array` a `MF_Array2`. Rozhranie je identické v obidvoch prípadoch. Rovnaký je aj mechanizmus správy súkromného pamäťového poolu, kam si objekty tried ukladajú pridelené pamäťové bloky. Rozdiel prichádza až vo fáze prístupu k subelementom pamäťových blokov súkromného poolu. Subelementy v podstate znamenajú miesta v pamäti o danej veľkosti v rámci jedného pamäťového bloku, ku ktorým má objekt triedy prístup pomocou ukazovateľov. Veľkosť subelementov je daná veľkosťou dátového typu zadaného ako parameter šablóny pri inicializácii dátovej štruktúry. Napríklad, blok pamäte veľkosti 512B je možné rozdeliť na 128 subelementov o veľkosti 4B, čomu zodpovedá napríklad dátový typ `Integer`. Subelement je teda len naším abstraktným označením miesta v pamäti a nejedná sa o ďalšiu štruktúru, alebo ďalšiu premennú triedy. Z úplne najjednoduchšieho pohľadu sa jedná o obyčajnú aritmetiku ukazovateľov v rámci jedného pamäťového bloku, respektíve v rámci súkromného pamäťového poolu.

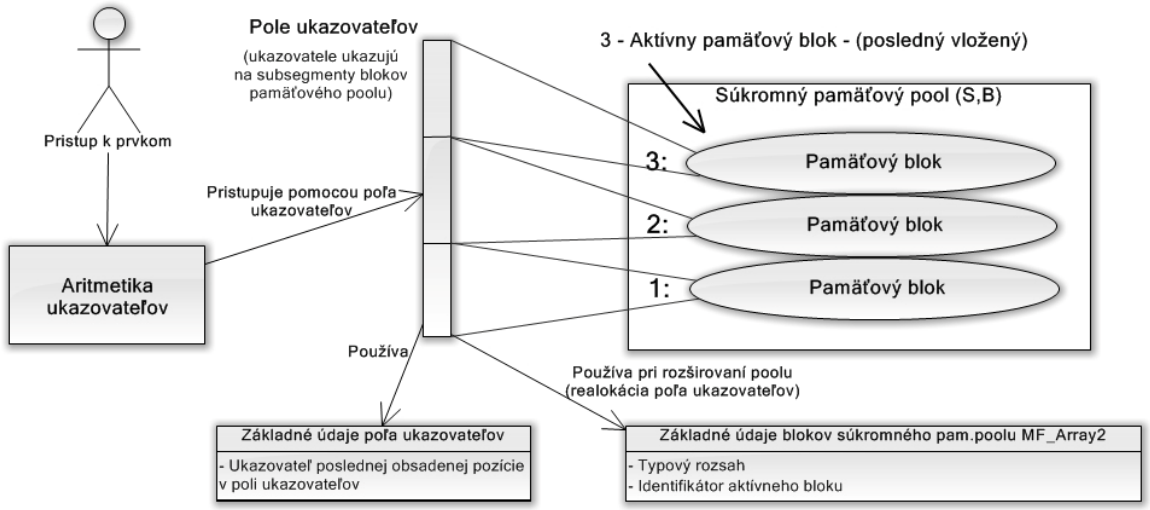
V prípade `MF_Array` sa pri inicializácii objektu inicializuje aj ukazovateľ daného typu na základe parametru šablóny, ktorý ukazuje na začiatok prvého pamäťového bloku a za aktuálny pamäťový blok sa označí blok prvý. Postupne pridávané prvky sú pomocou ukazovateľa ukladané priamo na konkrétne miesta v pamäti v rámci aktuálneho pamäťového bloku. Ak dôjde k situácii, že aktuálny pamäťový blok je plný, objekt dátovej štruktúry si požiada o ďalší voľný blok pamäte, identifikátor aktívneho bloku sa inkrementuje o jedničku a pozičný ukazovateľ sa nastaví na začiatok druhého bloku (obrázok 10).

V druhom prípade sa v `MF_Array2` inicializujú rovnaké ukazovatele, ako u predchádzajúceho `MF_Array`. Okrem toho sa vyalokuje na halde nové pole ukazovateľov potrebnej veľkosti, ktoré sa inicializujú, ako kópie ukazateľov na subelementy prvého pamäťového bloku súkromného poolu. Aritmetika ukazovateľov je tak realizovaná len v rámci jednej štruktúry poľa ukazovateľov. O ďalší blok pamäte si dátová štruktúra rovnako, ako v prechádzajúcom prípade, požiada v prípade naplnenia aktuálneho pamäťového bloku. V rámci rozšírenia pamäťového poolu sa realokuje i pole ukazovateľov dátového typu šablóny (obrázok 10).

Dátové štruktúry rámca typu zásobník: `MF_Stack`, `MF_Stack2` sú založené na podobných mechanizmoch a algoritmoch ako `MF_Array` a `MF_Array2`, takže rozdiely v prístupe k subelementom pamäťových blokov sú si veľmi podobné aj v prípade realizácie zásobníkov. Dátová štruktúra typu zoznam má v rámci len jedno zastúpenie, nakoľko jej fyzická štruktúra a práca s pamäťovými blokmi sa trochu odlišuje od poľa a zásobníku a počas jej implementácie sa nenaskytla potreba realizácie dvoch riešení. `MF_List` zoznam je štandardne zložený z uzlov, ktoré sú v implementácii rámca pretypovávané priamo do pamäťových blokov pridelených správcom. To znamená, že aj počas použitia dátovej štruktúry `MF_List` nedochádza k žiadnej „pomalej“ alokácii a uvoľňovaniu na halde pomocou `new/delete`, ale všetky operácie prebiehajú v pamäti rámca.



a) MF_Array



b) MF_Array2

Obrázok 10: Ukážka rozdielneho prístupu k prvkom dátových štruktúr rámca `MF_Array` (a) a `MF_Array2` (b). Dátová štruktúra poľa `MF_Array` pristupuje k subsegmentom pamäťových blokov súkromného poolu priamo, za pomoci premenných uvedených na hornom obrázku. `MF_Array2` pristupuje k subsegmentom prostrednictvom poľa ukazovateľov, kde ukazovatele odkazujú na subsegmenty blokov pamäťového poolu.

8.2.3 Modul testov

Modul testov obsahuje triedy, ktoré slúžia k prevedeniu navrhovaných testov, ktorým sme sa začali venovať už pri návrhu. Modul je skutočne jednoduchý a obsahuje len 3 triedy, kde prvá trieda `TestObject` vytvára objekty o veľkosti 48B konštruktorom bez parametrov, druhá trieda `Test` prevádza výkonnostné testy a tretia trieda `TestCorrectness` prevádza test korektnosti.

Trieda `Test` obsahuje niekoľko metód reprezentujúcich jednotlivé testy, ktoré sme si v krátkosti spomenuli už v návrhu. Trieda `TestCorrectness` testuje primárne spoľahlivosť správcu pamäte rámca. Za týmto účelom si objekt triedy deklaruje a inicializuje rámec veľkosti 200 pamäťových blokov v režime zamietnutého rozšírenia pamäťových poolov. Rámec tak v prípade nedostatku pamäťových blokov nealokuje ďalšiu pamäť z haldy, ale vracia prázdnu hodnotu. V ďalšom kroku si objekt vytvorí 5 vlákien, každé s vlastnou metódou vlákna, ktoré postupne žiadajú o pamäťové bloky správcu pamäte a následne bloky uvoľňujú. Po pridelení nového bloku pamäte si každé vlákno kontroluje, či mu nebol pridelený blok, ktorého vlastníkom je už iné vlákno. Vlastníctvo bloku je realizované príznakom ID vlákna nachádzajúcom sa v každom pamäťovom bloku, v testovacom móde rámca. Vlákno, ktorému je pri požiadavke o blok pamäte vrátená prázdna hodnota, počká na uvolnenie bloku z iného vlákna a o blok opätovne zažiada. Bližšie sa budeme k prevedeniu a výsledkom testov venovať v poslednej 9.kapitole.

9 Testovanie

Základné a štandardné úrovne testovania sú definované ako súbor testov, ktorými je softvér kontrolovaný na danom stupni podrobnosti. Podľa toho v akej fáze a s akým časovým odstupom od napísania kódu sa testovanie prevádza, ho delíme do piatich základných úrovní: *Testovanie programátorom* (*angl.* developer testing), *Testovanie komponent* alebo jednotiek (*angl.* unit testing), *Integračné testovanie* (*angl.* integration testing), *Systémové testovanie* (*angl.* system testing) a *Akceptačné testovanie* (*angl.* acceptance testing).

Procesy testovania námi vytvoreného rámca sa v istom smere odlišujú od bežných testov uživateľských aplikácii, nakoľko rámec pracuje výhradne v priestore hlavnej pamäte a bez akéhokoľvek grafického užívateľského rozhrania - GUI. Správnosť činností a procesov rámca v hlavnej pamäti je možné realizovať len manuálnou kontrolou, prípadne pomocou špecifických jednoduchých funkcií vytváraných v priebehu implementácie. Procesy testovania rámca môžeme rozdeliť z časového pohľadu ich realizácie do dvoch skupín:

- *Priebežné testy* – toto testovanie bolo prevádzané postupne počas implementácie rámca a zo základného súboru testov sem môžme zaradiť testovanie komponent a integračné testovanie.
- *Konečné testy* – úlohou tohto testovania je overenie a porovnanie výkonnosti rámca (tzv. benchmarky) a jeho spoľahlivosti.

9.1 Priebežné testy

Priebežné testy boli prevádzané postupne počas implementácie rámca, na úrovni jednotiek, komponent, modulov a následne na úrovni ich vzájomnej integrácie. Hlavným zámerom testov bolo overenie spoľahlivosti alokácie pamäťových blokov, ukladanie, upravovanie a prístup k premenným uloženým v pamäťových blokoch a pod.

U správcu pamäte sa jednalo primárne o kontrolu práce s pamäťovými blokmi, ich prideľovanie externým procesom a manipuláciu s pamäťovými blokmi priamo v externých procesoch. Súčasťou testovania bola taktiež kontrola úplného uvolnenia pamäte, ktorá bola alokovaná počas inicializácie správcu pamäte. V module dátových štruktúr rámca sme taktiež testovali spoľahlivosť vkladania a prístupu k prvkom uloženým v dátových štruktúrach. Jednalo sa hlavne o opakovaný prístup k jednému alebo viacerým prvkom v priebehu rôznych rozšírení a realokácii dátových štruktúr.

Priebežné testy sme prevádzali vo väčšine prípadov inšpekciou kódu, krokovaním, pomocou výpisov na konzolu, prípadne pomocou jednoduchých kontrolných funkcií.

9.2 Konečné testy

Konečné testy sme si už v krátkosti spomenuli v predchádzajúcich kapitolách. Testy sú primárne zamerané na overenie a porovnanie výkonnosti a spoľahlivosti nášho rámca. Výkonnostné testy, tzv. benchmarky boli prevádzané a porovnávané s bežne dostupnými metódami alokácie a uvolňovania pamäte v C++ a dátovými štruktúrami dostupnými v knižniciach CRT. V tejto kapitole si podrobne rozoberieme priebeh a výsledky testov a každý test v jeho závere zhodnotíme.

Konečné testy môžeme ďalej rozdeliť na výkonnostné testy (tzv. benchmarky) a testy spoľahlivosti (*angl.* correctness test).

9.2.1 Výkonnostné testy

Úlohou výkonnostných testov je overiť a porovnať výkon námi implementovaného rámca, v porovnaní so štandardnými spôsobmi práce s pamäťou v C++, ako je alokácia a uvoľňovanie pamäte na halde pomocou `new/delete`, využitie dátových štruktúr knižnice `STD` apod. U výkonnostných testov sme previedli celkom 5 testov, kde prvé dva testy boli realizované nad dátovými štruktúrami typu pole, tretí a štvrtý test boli realizované nad štruktúrou zásobníku a posledný piaty test bol realizovaný nad dátovou štruktúrou spojového zoznamu. Výsledkami výkonnostných testov sú zpriemerované časy a hodnoty spotreby pamäte. Konkrétne sme u testov zaznamenávali: *process time*, *user time, kernel time* pomocou triedy `cTimer`. Celkom sme prevádzali u každého testu 15 opakovaní a výsledky zpriemerovali do výslednej hodnoty. Ako referenčný objekt prvkov dátových štruktúr bol vo všetkých prípadoch použitý objekt typu `TestObject` veľkosti 48B.

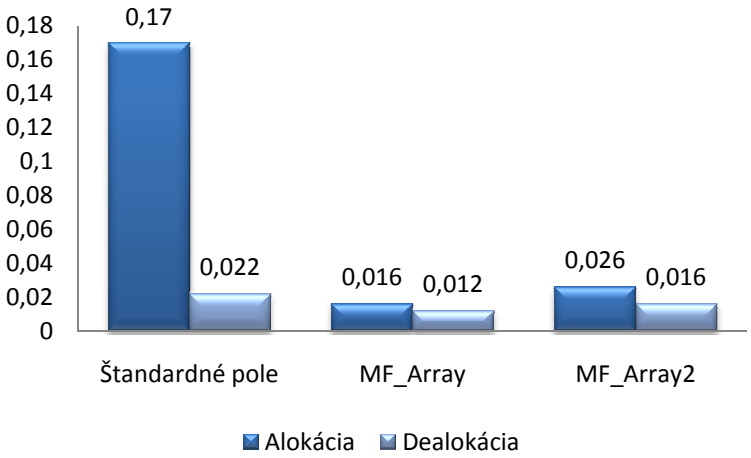
Testy sme prevádzali na PC s operačným systémom MS Windows Vista Home Premium 32-bit, procesor Intel Core 2 Duo T7500 (2200 MHz, 800 MHz FSB, 4 MB L2 Cache), operačná pamäť 2 GB DDR2.

Výkonnostný test 1 – Test rýchlosti alokácie a dealokácie polí
Prvým testom je test rýchlosti alokácie a dealokácie väčšieho množstva jednotlivých polí. Pri teste boli použité celkom tri typy dátovej štruktúry poľa:

- *Štandardné pole* – pole alokované na halde štandardným spôsobom pomocou príkazov `new/delete`.
- *MF_Array* – prvý typ dátovej štruktúry rámca. Aj samotná alokácia objektov prebiehala v pamäťových blokoch rámca bez použitia `new/delete`.
- *MF_Array2* – druhý typ dátovej štruktúry s odlišným algoritmom práce s pamäťovými blokmi. Aj v tomto prípade prebiehala alokácia objektov v pamäťových blokoch rámca, ako v predchádzajúcom prípade.

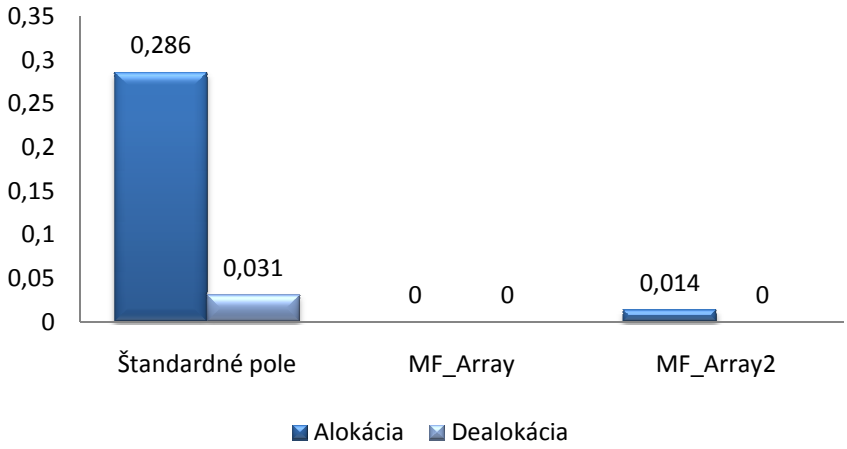
Hneď u prvého testu sme narazili na problém s uvoľňovaním pamäte v OS Windows, u bežných polí alokovaných na halde pomocou `new` a `delete`. OS Windows, respektíve správca haldy, totiž neuvolní všetku pamäť pri finalizácii poľa, nakoľko počíta s jej ďalším využitím pri behu aplikácie – viac v kapitole 9.3. V prvom teste nám pamäť pridelená poliam, alokovaným a dealokovaným príkazmi `new` a `delete`, zostávala takmer všetka neuvolnená. Z tohto dôvodu sme sa rozhodli vytvoriť alternatívu prvého testu, ktorá dokazuje že OS Windows si skutočne ponecháva istý počet bytov každého poľa v pamäti aj po jeho uvolnení.

V prvej variante testu sme u všetkých troch typov polí postupne alokovali a dealokovali 30 000 polí daného typu, pričom sme merali vždy samostatne čas alokácie a čas dealokácie. Veľkosť štandardných polí a dátové štruktúry `MF_Array` a `MF_Array2` sme sa snažili voliť a nastaviť tak, aby veľkosť alokovanej pamäte bola u všetkých typov rovnaká. Výsledky testu sú znázornené tabuľkou v prílohe C.1 a graficky na obrázku 11.



Obrázok 11: Výsledky 1.výkonnostného testu rýchlosti alokácie a dealokácie polí, zobrazené pomocou stĺpcového grafu. Veličinou vertikálnej osy je user time.

U alternatívneho testu sme naopak alokovali a dealokovali len 10 polí u každého typu, s tým že každé pole malo mnohonásobne väčšiu veľkosť. U štandardného poľa sme volili veľkosť poľa `1 000 000` a u dátových štruktúr rámca sme nastavili blok správcu pamäte až na `48 000 000B`. Výsledky testu sa taktiež nachádzajú v prílohe C.1, prípadne na obrázku 12.



Obrázok 12: Výsledky alterntívneho 1.výkonnostného testu rýchlosti alokácie a dealokácie polí, zobrazené pomocou stĺpcového grafu. Veličinou vertikálnej osy je user time.

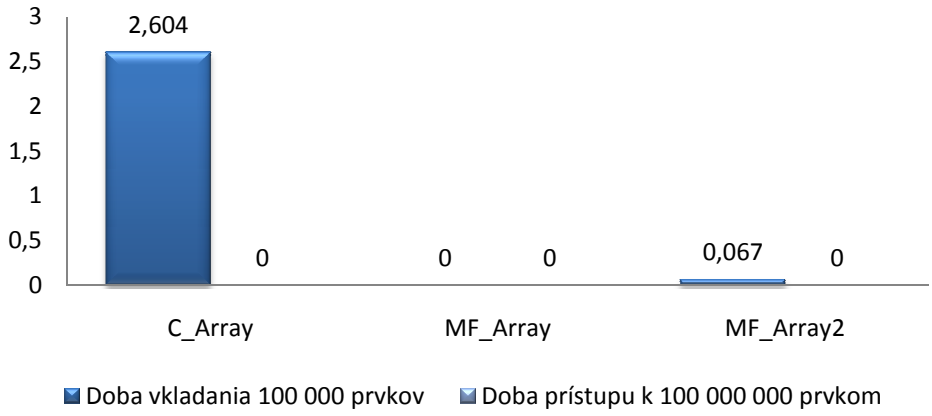
Záver testu: Alokácia i dealokácia polí prebiehala najdlhšie u štandardných polí alokovaných pomocou `new/delete`. Dátové štruktúry rámca `MF_Array` a `MF_Array2` boli pri alokácii výrazne rýchlejšie oproti štandardnému poľu. Rýchlosť dealokácie bola dosť podobná u všetkých troch typov polí. `MF_Array` mal v porovnaní s `MF_Array2` trochu lepšie výsledky, čo sa dalo očakávať nakoľko `MF_Array2` si pri inicializácii vytvára samostatné pole ukazovateľov, do ktorého musí následne namapovať pamäť jeho súkromného pamäťového poolu. Tento algoritmus má teda vyššie nároky pri inicializácii aj rušení poľa.

Výsledky obidvoch testov sa zdajú byť veľmi podobné, hlavný rozdiel je však v uvoľňovaní pamäte, ktoré je vidieť z tabuliek spotreby pamäte v prílohe C.1. U prvého testu sa po uvolnení štadardného poľa nevrátila operačnému systému takmer žiadna pamäť, a naopak v druhom prípade sa mu vrátila takmer všetka. Problém vychádza z algoritmov implementovaného správcu haldy v konkrétnom operačnom systéme. Podrobnejšie sa problému venujeme v kapitole 9.3

Výkonnostný test 2 – Test rýchlosti vkladania a prístupu k prvkom poľa
Druhým výkonnostným testom bol test rýchlosti vloženia väčšieho množstva prvkov do jednotlivých typov polí a rýchlosť prístupu k vloženým prvkom. Pri teste boli použité celkom tri typy dátovej štruktúry poľa:

- *C_Array* – dynamické pole využívajúce metódy `malloc`, `memcpy` implementované pre účel výkonového testu. Trieda je redukovaná len na základné procesy potrebné k prevedeniu testu.
- *MF_Array* - prvý typ dátovej štruktúry rámca.
- *MF_Array2* - druhý typ dátovej štruktúry s odlišným algoritmom práce s pamäťovými blokmi.

U každého typu poľa sme vytvorili jeden objekt triedy a do každého poľa sme vložili 100 000 prvkov typu `TestObject` o veľkosti 48B, pričom sme merali čas od vloženia prvého prvku po vloženie prvku posledného. V druhej časti testu sme náhodne pristupovali k 100 000 000 prvkom. Metóda testu si pred realizovaním prístupu k prvok vygeneruje pole typu `INT` s hodnotami v rozsahu od 0 do 99 000. Následne sú tieto hodnoty použité, ako náhodné pozície v poli. Prístup prebieha v dvoch cykloch, kde vonkajší cyklu beží 1 000 krát a vnútorný cyklus s „random“ poľom beží 100 000 krát (1 000 x 100 000 = 100 000 000). Výsledky testov sú znázornené tabuľkou v prílohe C.2 a graficky na obrázku 13.



Obrázok 13: Výsledky 2.výkonnostného testu rýchlosti vkladania a prístupu k prvkom jednotlivých polí, zobrazené pomocou stĺpcového grafu. Veličinou vertikálnej osy je user time.

Záver testu: Rovnako, ako u prvého testu je vkladanie prvkov do dátových štruktúr rámca (`MF_Array`, `MF_Array2`) mnohonásobne rýchlejšie v porovnaní s bežným riešením dyna-

mického poľa. Štruktúra MF_Array mala dokonca nulovú časovú réžiu vkladania prvkov. U všetkých troch typov polí dochádzalo k priebežnému rozširovaniu pamäťových prostriedkov v závislosti na množstve vložených prvkov. U C_Array bolo samozrejme možné vyalokovať na počiatku pole s veľkosťou 100 000 prvkov a k jeho ďalšiemu rozširovaniu nemuselo dochádzať. Aby sme však ukázali výhody dátových štruktúr rámca oproti dynamickému poľu, inicializovali sme pole C_Array na 170 prvkov a každá realokácia rozširovala pole o ďalších 170 prvkov. 170 prvkov zodpovedá veľkosti jedného pamäťového bloku B vzhľadom k dátovému typu TestObject veľkosti 48B, kde *170 * 48B = 8160B*. Pamäť sa teda u všetkých typov polí rozširuje cca každých 170 vložených prvkov. Nárast času vkladania prvkov u štandardného dynamického poľa C_Array (obrázok 13) spôsobila neefektívna realokácia, kedy sa pomocou malloc vyalokuje na halde nové pole o väčšej veľkosti, menšie pole sa pomocou memcpy prekopíruje do nového väčšieho poľa a na záver sa predchádzajúce menšie pole z haldy uvolní.

Trochu nečakaným výsledkom bol náhodný prístup k prvkom jednotlivých polí, ktorý mal u všetkých typov polí nulovú réžiu. Nulovú réžiu spôsobila L2 cache pamäť, ktorej veľkosť 4MB pokryla takmer celé pole, v ktorom sa v danej chvíli náhodne pristupovalo k prvkom. Procesor tak nemusel pristupovať k hlavnej pamäti, ale pracoval z väčšej časti len s pamäťou cache.

Výkonnostný test 3 – Test rýchlosti alokácie a dealokácie zásobníkov

Test rýchlosti alokácie a dealokácie jednotlivých typov zásobníkov je tretím výkonnostným testom, ktorý sme prevádzali. Pri teste boli použité celkom tri typy zásobníkov:

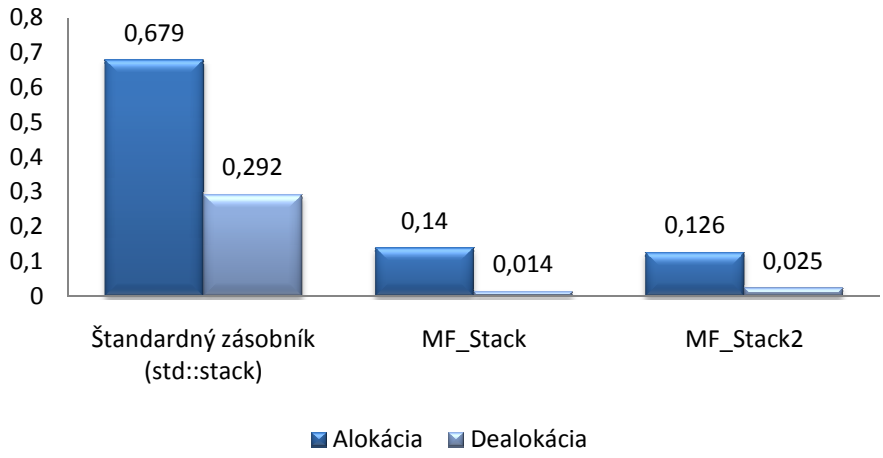
- *Štandardný zásobník* – knižnica Stack dostupná v bežnej implementácii behového prostredia C++.
- *MF_Stack* – prvý typ dátovej štruktúry rámca. Rovnako, ako u dátových štruktúr MF_Array a MF_Array2, boli objekty MF_Stack alokované v pamäťových blokoch rámca bez použitia new/delete.
- *MF_Stack2* – druhý typ dátovej štruktúry s odlišným algoritmom práce s pamäťovými blokmi.

U všetkých troch typoch zásobníku sme postupne alokovali a dealokovali 30 000 zásobníkov a do každého zásobníku sme počas alokácie vložili 150 prvkov. Doplnenie testu o vkladanie prvkov sme volili z toho dôvodu, že alokácia prázdnych zásobníkov bola vo všetkých prípadoch veľmi rýchla, s nulovou časovou réžiou.

Záver testu: Na základe výsledných časov testu, zobrazených v prílohe C.3 a na obrázku 14, je opäť vidieť že dátové štruktúry zásobníku rámca pracujú pri alokácii a dealokácii objektov rýchlejšie v porovnaní so štandardným zásobníkom dostupným v knižniciach C++.

Keďže sme v tomto teste alokovali štandardný zásobník s počiatočnou veľkosťou 0 a postupne sme zásobník naplnili 150 prvkami, previedli sme ešte jeden test, kedy sme štandardný zásobník alokovali na počiatočnú veľkosť 170 prvkov objektu TestObject pomocou kontajneru deque. Veľkosť 170 prvkov zodpovedá jednému BIG bloku, ktorý majú k dipozícii dátové štruktúry MF_Stack, MF_Stack2 pri ich inicializácii (8192B / 48B = 170). Štandardný kontajner zásobníku v C++ nepodporuje explicitné nastavenie veľkosti zásobníku počas jeho incializácie, ale len možnosť inicializácie kopírovaním zo štruktúr deque a vector. Výsledky testu tu nebudeme uvádzať, nakoľko alokácia zásobníkov kopírovaním z deque bola veľmi

pomalá a dosahovala priemernú dobu alokácie cca 1,8s a dealokácie cca 0,7s, čo je s nulovou réžiou `MF_Stack` a `MF_Stack2` neporovnateľné. Inicializácia štandardných zásobníkov kopírovaním z `deque` bola dokonca pomalšia, ako samotný 3.test alokácie prázdnych štandardných zásobníkov s následným vložením 150 prvkov typu `TestObject`.



Obrázok 14: Výsledky 3.výkonnostného testu rýchlosti alokácie a dealokácie zásobníkov, zobrazené pomocou stĺpcového grafu. Veličinou vertikálnej osy je user time.

Výkonnostný test 4 – Test rýchlosti vloženia a výberu prvkov z jednotlivých zásobníkov

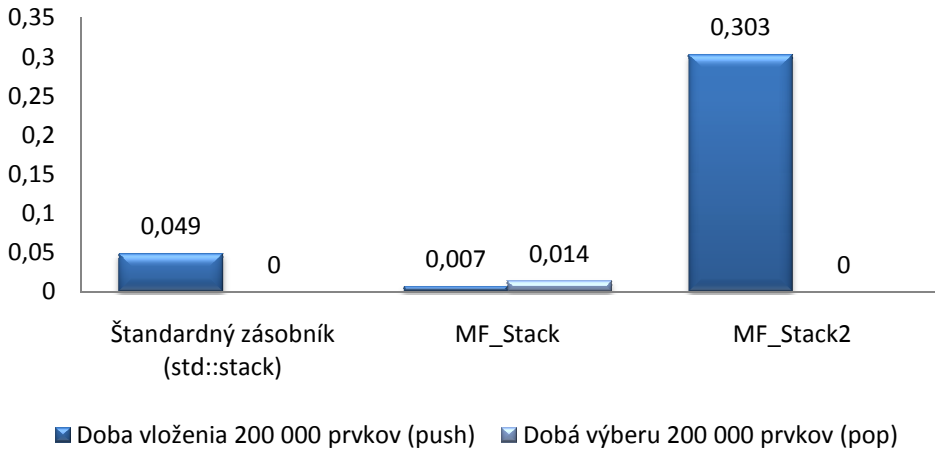
Štvrtým výkonnostným testom je test vloženia (metóda `push`) a výberu (metóda `pop`) prvkov z jednotlivých zásobníkov. Test sme realizovali, rovnako ako test 3, na troch typoch zásobníkov:

- *Štandardný zásobník* – knižnica `Stack` dostupná v bežnej implementácii behového prostredia C++.
- *MF_Stack* – prvý typ dátovej štruktúry rámca. Rovnako, ako u dátových štruktúr `MF_Array a MF_Array2`, boli objekty `MF_Stack` alokované v pamäťových blokoch rámca bez použitia `new/delete`.
- *MF_Stack2* – druhý typ dátovej štruktúry s odlišným algoritmom práce s pamäťovými blokmi.

Počas testu sme vyalokovali jeden objekt pre každý typ zásobníku a do každého zásobníku sme pridali 200 000 prvkov typu `TestObject`. Následne sme z každého zásobníku tieto vložené prvky vybrali. U jednotlivých typov zásobníku sme samostatne merali čas vloženia prvkov do zásobníku a čas ich výberu. Štandardný zásobník bol inicializovaný na nulovú veľkosť nakoľko kontajner stack v C++ nepodporuje explicitnú inicializáciu veľkosti zásobníku. Túto vlastnosť sme si už popísali v závere 3.výkonnostného testu.

Záver testu: Výsledky testu sú znázornené v prílohe C.4 a na obrázku 15. Z výsledkov je vidieť, že proces vkladania prebieha najrýchlejšie v dátovej štruktúre `MF_Stack` a najpomalšie v dátovej štruktúre `MF_Stack2`, čo je spôsobené neefektívnou priebežnou realokáciou poľa ukazovateľov. Väčšiu časovú réžiu sme očakávali už pri návrhu algoritmu dátovej štruktúry `MF_Stack2`.

Proces výberu prvkov prebieha najrýchlejšie, dokonca s nulovou časovou réžiou, v štandardnom zásobníku C++. Rýchlosť výberu u štandardného zásobníku je daná tým, že štandardný zásobník po odobraní prvkov nevracia pridelenú pamäť späť operačnému systému. Po odobrani 200 000 prvkov tak ostáva pamäťová réžia štandardného zásobníku nezmenená. Dátové štruktúry `MF_Stack` a `MF_Stack2` pracujú s pamäťou efektívnejšie a nepotrebnú pamäť vracajú späť správcovi pamäte, čo má za následok určitú časovú réžiu viditeľnú vo výsledkoch testu.



Obrázok 15: Výsledky 4.výkonnostného testu vloženia 200 000 prvkov (push) do jednotlivých zásobníkov a ich výber (pop).

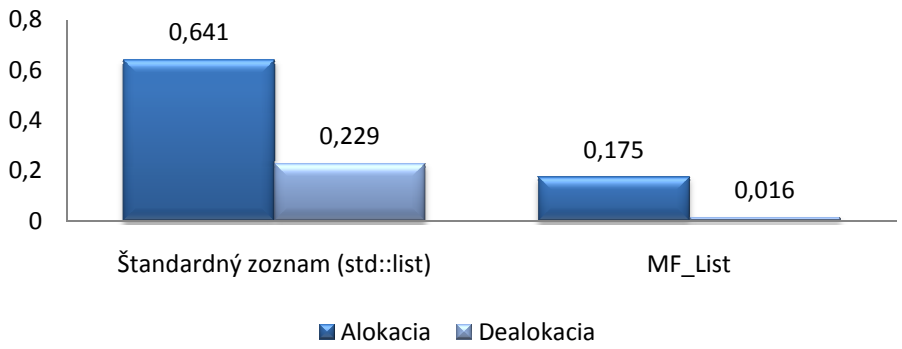
Výkonnostný test 5 – Test rýchlosti alokácie a dealokácie zoznamov

Ďalším výkonnostným testom je test rýchlosti alokácie a dealokácie zoznamov. Test sa líši od predchádzajúcich prípadov nakoľko do testu vstupujú len dve dátové štruktúry. Riešenie rámca totiž obsahuje len jeden typ dátovej štruktúry zoznamu.

- *Štandardný zoznam* – knižnica `List` dostupná v bežnej implementácii behového prostredia C++.
- *MF_List* – dátová štruktúra rámca typu zoznam. Pri svojej činnosti využíva výhradne pamäťové bloky pridelené správcom pamäte rámca.

V teste sme postupne u každého typu zoznamu vyalokovali a dealokovali 30 000 zoznamov a do každého zoznamu sme vložili 150 prvkov veľkosti 48B. Vloženie 150 prvkov sme volili opäť za účelom zvýšenia zložitosti procesu, nakoľko alokácia prázdnych zásobníkov bola veľmi rýchla, rovnako, ako tomu bolo u zásobníkov. Počas testovania sme samostatne merali čas ich alokácie a dealokácie.

Záver testu: Z výsledkov testu, zobrazených v prílohe C.5 a obrázku 16, je opäť jednoznačne vidieť, že rýchlosť alokácie i dealokácie väčšieho množstva zoznamov prebieha rýchlejšie s použitím dátovej štruktúry rámca.



Obrázok 16: Výsledky 5.výkonnostného testu alokácie a dealokácie zoznamov, zobrazené pomocou stĺpcového grafu. Veličinou vertikálnej osy je user time.

9.2.2 Test korektnosti

Úlohou testu korektnosti (*angl.* correctness test) je overenie a preukázanie spoľahlivosti behu rámca, konkrétne modulu správy pamäte, počas súbežného prístupu viacerých externých procesov. V podstate sa jedná o situáciu kedy bude v jednom okamžiku žiadať o pamäťový blok niekoľko externých procesov. Návrh a implementácia správcu pamäte s touto situáciou počíta, a implementácia tento stav rieši. Výsledné riešenie je však potrebné overiť, čo bude úlohou tohto testu.

V kapitole implementácie sme sa dozvedeli, že test korektnosti prevádza trieda `TestCorrectness`, ktorá vytvára 5 samostatných vlákien realizovaných 4 metódami. Trieda `TestCorrectness` si inicializuje správcu pamäte o malej veľkosti 200 blokov v režime zamietnutého rozšírenia pamäťových poolov. To znamená, že správca pamäte má statickú veľkosť 200 blokov a pri úplnom vypráznení jeho pamäťového poolu nedôjde k alokácii ďalších blokov, ale správca pamäte vracia hodnotu `NULL`. Následne testovací objekt vytvorí 5 vlákien, ktoré súbežne žiadajú o pamäťové bloky a pamäťové bloky vracajú späť správcovi. Každá zo štyroch metód vlákien žiada a uvoľňuje pamäťové bloky iným algoritmom, takže požiadavky a uvolnenia nie sú úplne identické. Po pridelení pamäťového bloku, je úlohou vlákna skontrolovať pamäťový blok, či náhodou nedošlo k priradeniu bloku, ktorého vlastníkom je už iné vlákno. Pokiaľ nastane situácia, že vláknu bol pridelený blok, ktorého vlastníkom je vlákno iné, vypíše túto skutočnosť do konzoly. V prípade, že je pamäťový pool správcu pamäte prázdny a správca vráti namiesto voľného bloku hodnotu `NULL`, vlákno počká na uvolnenie ďalšieho bloku z iného vlákna. Test korektnosti je možné spustiť v dvoch módoch, ako *výkonnostný test korektnosti* kedy vlákna bežia bez prestávok a v šetrnejšom *móde s uspaním vlákien*. V šetrnejšom móde sú vlákna na krátku dobu cca od 1000 do 3000 ms úspávané počas činnosti vlákna, čo má vplyv na zníženie záťaže mikroprocesoru. Naopak, výkonnostný test nám umožňuje otestovať správnosť činnosti pamäťového správcu vo vysokom zaťažení, kedy dochádza naozaj k častým konfliktom súbežného prístupu ku kritickej časti správcu pamäte.

Test korektnosti sme prevádzali celkom v 10-tich opakovaniach, následovne:

- *Výkonnostný test korektnosti* – 5 opakovaní, kde každé opakovanie bežalo približne 5-8 minút. Limit ukončenia testu nevychádzal z jeho stanovenia, ale z teploty procesoru. Test sme ukončovali, keď procesor dosiahol teplotu 90 stupňov.

- *Šetrný test korektnosti* – 5 opakovaní, kde každé opakovanie (spustenie testu) bežalo 15 minút. Pauzy v behu vlákien znížili zaťaženie procesoru, takže 15 minutový test nebol pre procesor moc zaťažujúci.

Výsledky testu: Vo všetkých 10-tich opakovaniach nedošlo ani raz k situáciu, kedy by bol vláknu pridelený blok pamäte, ktorého vlastníkom je iné vlákno. Test korektnosti tak môžme vyhodnotiť za úspešný a modul správy pamäte pracuje so 100% spoľahlivosťou.

9.3 Problémy, na ktoré sme narazili pri testovaní

Počas testovania rámca, konkrétne u výkonových testov alokácie a dealokácie štandardných polí, sme narazili na problém s uvolňovaním pamäte. Pri testovaní alokácie a dealokácie dvojrozmerného poľa typu `TestObject` sme zistili, že systémový správca pamäte pri dealokácií neuvolní všetku pamäť operačnému systému. Aby sme vylúčili možný únik pamäte v riešení námi implementovaného rámca, vytvorili sme novú konzolovú aplikáciu, ktorá obsahuje len dve metódy pre alokáciu a dealokáciu jednorozmerného a dvojrozmerného poľa o približnej veľkosti *500MB*. Prvá metóda alokuje jednorozmerné pole typu `Integer` veľkosti *(512*512*512)* a druhá metóda alokuje dvojrozmerné pole typu `Integer`, kde prvý rozmer je veľkosti *(512*512)* a druhý rozmer veľkosti *512*. Dvojrozmerné pole tak alokuje *512*512=262 142* blokov o veľkosti *2048B*. Zdrojový kód testu je dostupný na priloženom CD. Rovnako, ako v testoch rámca, aj v tomto prípade zostáva po dealokácii časť pamäte naďalej pridelená procesom aplikácie. Zaujímavosťou je, že u jednorozmerného poľa rovnakej veľkosti *500MB* je procesu pridelená pamäť po dealokácii, až na pár bytov, kompletne vrátená operačnému systému.

Ďalším kontrolným, respektíve vylučovacím krokom, bolo spustenie jednoduchého testu pod iným operačným systémom. Pripomínam len, že rámec sme primárne testovali v 32-bitovom operačnom systéme *MS Windows Vista Home Premium*. Za účelom overenia chovania v inom systéme sme test previedli v OS *MS Windows 7 Professional 32-bit*, kde u dvojrozmerného poľa po jeho dealokácii správca pamäte taktiež nevrátil všetku pamäť späť operačnému systému. Veľkosť zbytkovej pamäte bola však nižšia, v porovnaní so systémom *MS Windows Vista*. Test sme previedli celkom 10 krát v každom operačnom systéme a stav pamäte sme sledovali procesnou utilitou *VMmap v3.1*, ktorá slúži k analýze virtuálnej a fyzickej pamäti. Výsledky testu sú zobrazené na obrázku 17 a v prílohe C.6.

Z reprezentácie pamäte v utilite *VMmap* je očividné, že operačný systém pri vytvorení dvojrozmerného poľa vyalokuje niekoľko pamäťových blokov haldy približnej veľkosti 16 000 KB, ďalšie členenie blokov nie je viditeľné z rozhrania *VMmap*, členenie by však malo zodpovedať bucketom uvedeným v prílohe A.1. Po dealokácii poľa však zostáva časť pamäťových blokov, veľkosti 4KB v *OS Windows* 7 a 4KB a 260KB v *OS Windows Vista*, naďalej pridelená procesu – *Committed* a tieto bloky samozrejme nie sú vrátené operačnému systému. Pozitívne je, že po opakovanej alokácii a dealokácii dvojrozmerného poľa tým istým procesom nedochádza k ďalšiemu úniku pamäte a správca haldy opätovne využíva pamäťové bloky haldy vrátane blokov, ktoré zostali v stave *Committed*. U jednorozmerného poľa nedochádzalo k žiadnym únikom pamäte.

Z testu je očividné, že správca haldy jednotlivých operačných systémov má menší problém s alokáciou, dealokáciou a s tým spojenou fragmentáciou pamäte u malých pamäťových blokov haldy. Pri alokácii väčšieho množstva malých blokov, tak stav spotrebovanej pamäte môže navodzovať fiktívny dojem úniku pamäte.

Vysvetlenie problému pri uvoľňovaní malých pamäťových blokov haldy sme sa snažili nájsť v rôznych literárnych aj elektronických zdrojoch. Bohužiaľ sa nám nepodarilo nájsť významný zdroj, ktorý by tento problém popisoval a vysvetľoval. Jediným vysvetlením, ktoré sa nám podarilo nájsť, je vyjadrenie podpory Microsoftu v dotaze uvedenom v diskusnom fóre Tech-Archive, kde Jeffrey Tan v krátkosti popisuje na základe internej konzultácie spomínaný problém s uvoľňovaním pamäte na halde. Podľa jeho slov problém spôsobuje rozdiel medzi priamym volaním funkcie `VirtualAlloc` a alokáciou z pamäťových blokov haldy. Správca pamäte totiž alokuje bloky veľkosti nad 512KB priamo volaním metódy `VirtualAlloc`, kdežto všetky ostatné menšie bloky sú alokované z blokov haldy, ktoré nemusia byť po dealokácii kompletne vrátené operačnému systému. Algoritmy správcu haldy môžu uvoľniť blok haldy buď úplne, alebo čiastočne, čo je závislé na rôznych vplyvoch, ako je napríklad fragmentácia haldy, použité algoritmy apod. [22].

Je vidieť, že algoritmy správcu haldy sa neustále vyvíjajú a zefektívňujú, avšak i naďalej sú prítomné určité nedostatky pri alokácii a uvoľňovaní pamäťových blokov, ktoré vo väčšine prípadov nespôsobujú žiadne problémy.

	MS Windows Vista Home Premium, 32-bit	**MS Windows 7 Professional, 32-bit**
Jednorozmerné pole		
Stav pamäte pred alokáciou	1416 KB	1496 KB
Stav pamäte po alokácii	526 740 KB	526 820 KB
Stav pamäte po dealokácii	1424 KB	1504 KB
Dvojrozmerné pole		
Stav pamäte pred alokáciou	1424 KB	1504 KB
Stav pamäte po alokácii	532 130 KB	532 212 KB
Stav pamäte po dealokácii	47 220 KB	4 904 KB

Obrázok 17: Porovnanie stavu pamäte počas alokácie a dealokácie jednorozmerného a dvojrozmerného poľa v operačných systémoch Windows Vista a Windows 7.

10 Záver

Hlavným cieľom diplomovej práce bolo preskúmanie problémov behového prostredia C++ a operačného systému pri práci s hlavnou pamäťou. Výsledkom práce je rámec pre vytváranie dátových štruktúr v hlavnej pamäti, ktorý pracuje rýchlejšie a efektívnejšie s pamäťou na aplikačnej úrovni a poskytuje programátorovi alternatívu k bežne dostupným metódam manuálnej alokácie a uvolňovania pamäte a k dátovým štruktúram dostupným v knižniciach C++.

Správa pamäte je skutočne veľmi zložitou problematikou, so širokou škálou možných riešení, či už na úrovni operačného systému, alebo na úrovni aplikačnej. Námi vytvorený rámec poskytuje programátorovi, výkonnejšiu alternatívu k štandardnej alokácii a uvolňovaniu pamäte na halde pomocou príkazov `new` a `delete` a alternatívu k štandardným dátovým štruktúram dostupným v knižniciach C++. Použitie modulu správy pamäte rámca vyžaduje jeho dostatočnú znalosť programátorom, nakoľko pri nesprávnej manipulácii s pamäťovými blokmi rámca môže dochádzať k neefektívnemu nakladaniu s pamäťou a k vzniku internej fragmentácie. Na druhú stranu, použitie modulu dátových štruktúr rámca je výrazne jednoduchšie a spoľahlivejšie, nakoľko si dátové štruktúry spravujú pamäťové bloky rámca sami. Výkonnostnými testami sme ukázali výhody nášho riešenia v porovnaní so štandardne dostupnými možnosťami z pohľadu rýchlosti alokácie a uvolňovania pamäte. Limitom a nevýhodou rámca je obmedzený rozsah veľkosti pamäťových blokov, ktoré sú v súčastnom riešení dostupné len v troch veľkostiach. Riešenie tohto nedostatku by mohlo byť podnetom pre ďalšie rozšírenie rámca, kde by sa realizoval samostatný modul ktorý by plne nahradzoval bežný prístup k pamäti v C++ pomocou `new` a `delete` a umožňoval by alokáciu pamäťových blokov rôznej veľkosti. Týmto rozšírením by mohol vzniknúť skutočne sofistikovaný rámec, ktorý by prostredníctvom dvoch režimov, s pevnou a dynamickou veľkosťou pamäťových blokov, poskytoval komplexné riešenie prístupu k pamäti na aplikačnej úrovni.

Táto diplomová práca mi umožnila rozšíriť si svoje znalosti a získať cenné skúsenosti v oblasti správy pamäte a to na všetkých troch úrovniach. Veľmi prínosné boli pre mňa informácie o správe pamäte v operačných systémoch MS Windows. Na záver by som chcel len dodať, že riešenie správy pamäti na aplikačnej úrovni, môže byť skutočne veľmi efektívnou metódou, ako zvýšiť výkon a rýchlosť behu výsledných aplikácii. Použitie rámca by sa malo značne prejaviť u väčších aplikácii, ktoré pracujú v hlavnej pamäti frekventovanejšie, s väčším množstvom dát a s viacvláknovým prístupom.

Zoznam použitej literatúry

[1] BENEŠ, Miroslav. *Správa paměti* [online]. 2003, 2003 [cit. 2012-03-15]. Dostupné z: http://www.cs.vsb.cz/benes/vyuka/pte/texty/pamet/

[2] LIČEV, Lačezar., *Architektura počítačů II*. FEI VŠB-TUO 1999.

[3] SOLČÁNY. Viliam. Správa pamäti: Úvod, metódy prideľovania pamäti. In: *Operačné systémy* [online]. 2011, 2011 [cit. 2012-03-16]. Dostupné z: http://osa.fiit.stuba.sk/os/slajdy2011/pp07mem.pdf

[4] TIŠNOVSKÝ, Pavel. Vyrovnávací paměti (cache), *Root.cz* [online]. 2008 [cit. 2012-03-19] Dostupné z: http://www.root.cz/clanky/vyrovnavaci-pameti-cache/

[5] HORÁK, Jaroslav. *Hardware: učebnice pro pokročilé.* 3., aktualiz. vyd. Brno: CP Books, 2005, 344 s. ISBN: 80-251-0647-0.

[6] BLUNDEN, Bill. *Memory management: algorithms and implementation in C/C++*. Plano, Tex.: Wordware Pub., c2003, 360 s. ISBN: 1-55622-347-1.

[7] HYDE, Randal. *The art of assembly language*, 2nd ed. San Francisco: No Starch Press, c2010, 732 s. ISBN: 978-1-59327—207-4.

[8] LIBERTY, Jesse. *Naučte se C++ za 21 dní*. 2. Aktualiz. vyd. Překlad Josef Pojsl, Karel Voráček. Brno: Computer Press, 2007, 796 s., ISBN: 978-80-251-1583-1.

[9] PARLANTE, Nick. *Pointers and Memory*. Stanford, 2000. Dostupné z: http://cslibrary.stanford.edu/102/PointersAndMemory.pdf

[10] Kernel. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2012-03-25]. Dostupné z: http://cs.wikipedia.org/wiki/Kernel

[11] RUSSINOVICH, Mark E. *Windows internals.* 5th ed. Washington, DC: Microsoft, c2009, 1232 s. ISBN: 978-0-7356-2530-3.

[12] BOLING, Douglas. Windows Embedded CE 6.0 Advanced Memory Management. *MSDN* [online]. 2007, 2010 [cit. 2012-03-30]. Dostupné z: http://msdn.microsoft.com/en-us/library/bb331824.aspx

[13] DING, Yu. *Heap Taichi: Exploiting memory allocation granularity in Heap-Spraying attack.*Austin [online], 2010 [cit. 2012-03-30]. Dostupné z: http://www.comp.nus.edu.sg/~liangzk/papers/acsac10.pdf

[14] DETLEFS, David. Al DOSSER a Benjamin ZORN. *Memory allocation costs in large C and C++ programs.* USA, 1994. Dostupné z: https://citeseerx.ist.psu.edu

[15] SHAFFER, Clifford A. *Memory Management Tutorial* [online], 2007, 2011 [cit. 2012-04-01], Dostupné z: http://research.cs.vt.edu/AVresearch/MMtutorial/

[16] TERRY, Luke. *Incremental Garbage Collection Using Method Specialisation.* London, 2010.Dostupné z: http://www.doc.ic.ac.uk/teaching/distinguished-projects/2010/l.terry.pdf. Final Report. Imperial College London

[17] Writing a Multithread Win32 Program. *MSDN* [online]. 2010, 2010 [cit. 2012-04-03], Dostupné z: <http://msdn.microsoft.com/cs-cz/library/z3x8b09y.aspx>

[18] CHOU, Dan. *A Quick and Versatile Synchronization Object* [online]. 1998 [cit. 2012-04-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/ms810428

[19] DVORSKÝ, Jiří. *Algoritmy I*, FEI VŠB-TUO 2008.

[20] WRÓBLEWSKI, Piotr. *Algoritmy: datové struktury a programovací techniky*. Vyd. 1. Preklad Marek Michálek, Bogdan Kiszka. Brno: Computer Press, 2004, 351 s. ISBN: 80-251-0343-9

[21] ISO/IEC 14882:2011. *Information technology– Programming languages – C++*. Switzerland: ISO copyright office, 2011.
Dostupné z: http://www.iso.org/iso/iso_catalogue.htm

[22] TAN, Jeffrey. HeapAlloc heap fragmentation. In: *Tech-archive.net* [online]. 2005 [cit. 2012-04-24]. Dostupné z: http://www.tech-archive.net/Archive/Development/microsoft.public.win32.programmer.kernel/2005-11/msg00228.html

Zoznam príloh

Príloha A – Zoznam tabuliek
Príloha B – Triedne diagramy
Príloha C – Výsledky testov
Príloha D – Priložené CD (zdrojové kódy rámca, programátorská príručka)

A Príloha – Zoznam tabuliek

A.1 Tabuľka 1

Prehľad bucketov OS Windows, ich granularita a rozsah [11].

Buckets	Granularita (byte)	Rozsah (byte)
1 – 32	8	1 – 256
33 – 48	16	257 – 512
49 – 64	32	513 – 1024
65 – 80	64	1025 – 2048
81 – 96	128	2049 – 4096
97 – 112	256	4097 – 8192
113 – 128	512	8195 – 16384

A.2 Tabuľka 2

Súhrrn vlastností sycnchronizačných objektov v OS Windows [18].

Názov	Realatívna rýchlosť	Synchronizácia naprieč procesmi	Počítanie zdrojov
Kritická sekcia	rýchla	nie	nie (exluzívny prístup)
Mutex	pomalá	áno	nie (exluzívny prístup)
Semafór	pomalá	áno	áno
Udalosť	pomalá	áno	áno *
Metered sekcia	rýchla	áno	áno

* Udalosti môžu byť použité pre počítanie zdrojov, ale bez informácie o ich počte pre programátora.

A.3 Tabuľka 3

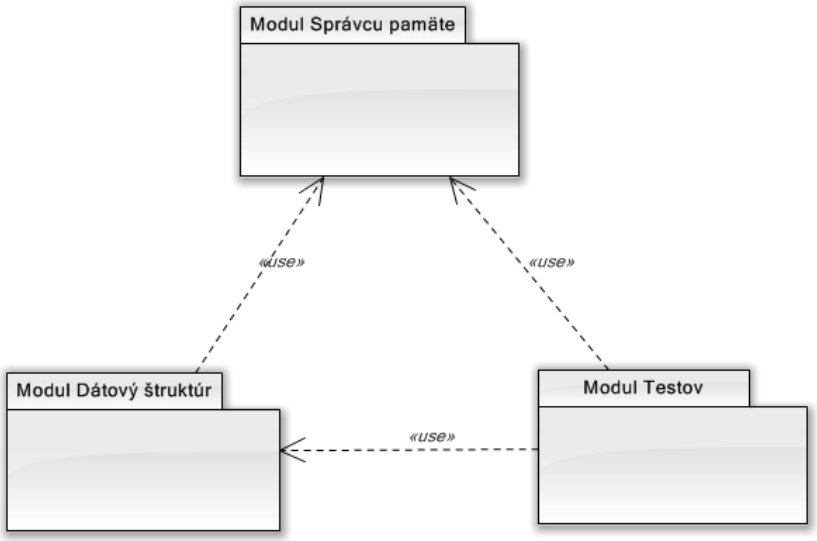
Navrhované rozhranie dátových štruktúr rámca (zoznam verejných metód dostupných užívateľovi rámca).

Dátová štruktúra rámca	Názov metódy	Poznámky
MF_Array (pole)	Add	Vloženie prvku na koniec poľa.
	operator[]	Preťaženie operátore, za účelom prístupu k prvkom poľa.
	Clear	Vymazanie prvkov poľa mimo počet predaný, ako parameter tejto metódy.
	ClearAll	Úplná finalizácia objektu.
	GetSize	Metóda vráti veľkosť poľa.
	GetCountOfUsedMemBlocks	Metóda vráti počet pamäťových blokov, ktoré sú vo vlastníctve objektu dát.štruktúry.
	GetSignBlockSize	Metóda vráti príznak veľkosti použitých pamäťových blokov objektom dát. štruktúry: S, B, T.
MF_Array2	Add	Vloženie prvku na koniec poľa.
	operator[]	Preťaženie operátore, za účelom prístupu k prvkom poľa.
	Remove	Metóda odstráni prvok na parametrom zadanej pozícii.
	Clear	Vymazanie prvkov poľa mimo počet predaný, ako parameter tejto metódy.
	ClearAll	Úplná finalizácia objektu.
	GetSize	Metóda vráti veľkosť poľa.
	GetCountOfUsedMemBlocks	Metóda vráti počet pamäťových blokov, ktoré sú vo vlastníctve objektu dát.štruktúry.
	GetSignBlockSize	Metóda vráti príznak veľkosti použitých pamäťových blokov objektom dát. štruktúry: S, B, T.
MF_Stack	Push	Metóda pridá prvok na vrchol zásobníku.
	Pop	Metóda odoberie prvok z vrchu zásobníku.
	Clear	Vymazanie prvkov zásobníku mimo počet predaný, ako parameter tejto metódy.
	ClearAll	Úplná finalizácia objektu.
	GetSize	Metóda vráti veľkosť poľa.
	GetCountOfUsedMemBlocks	Metóda vráti počet pamäťových blokov, ktoré sú vo vlastníctve objektu dát.štruktúry.
	GetSignBlockSize	Metóda vráti príznak veľkosti použitých pamäťových blokov objektom dát. štruktúry: S,

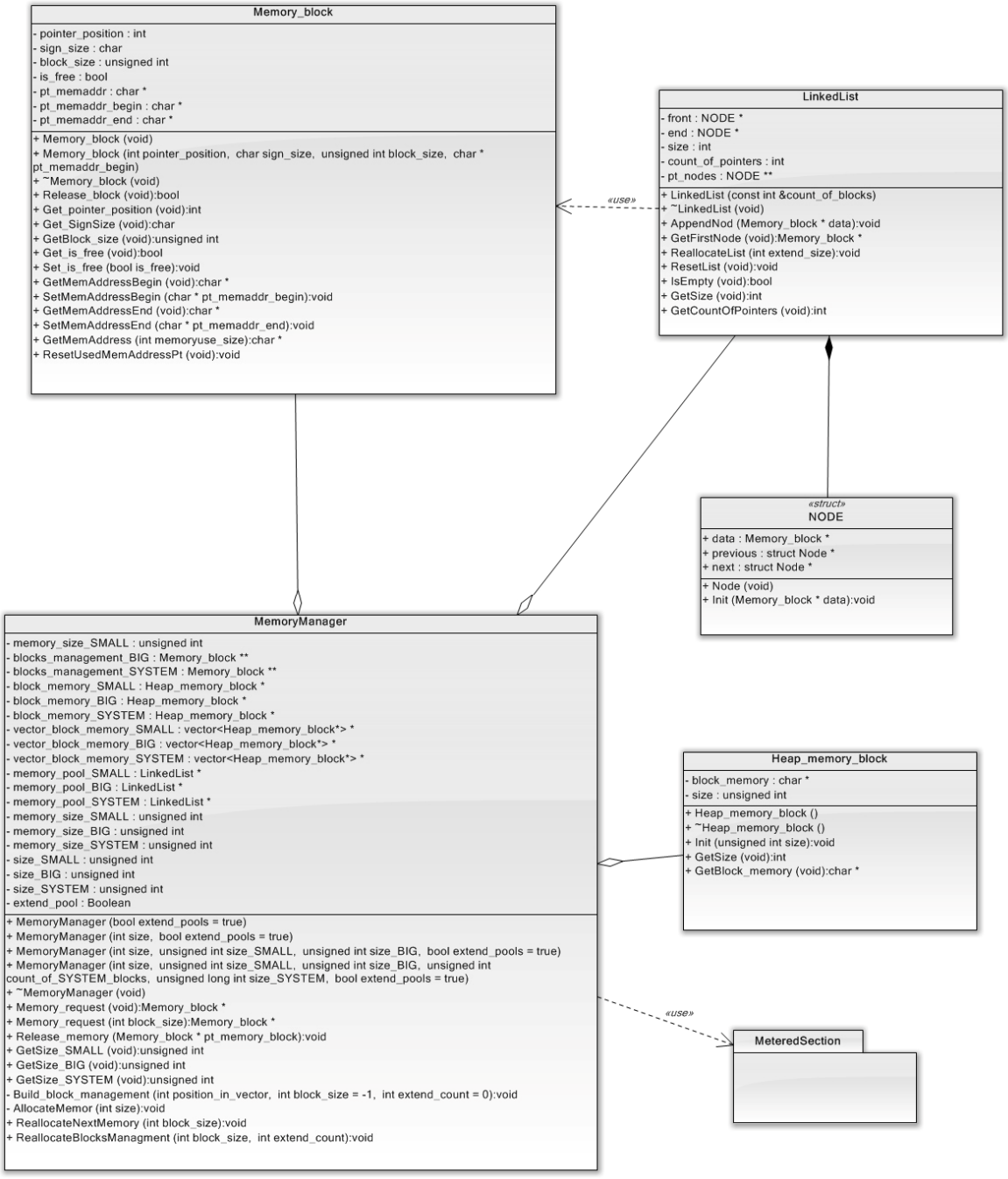
		B, T.
	IsEmpty	Metóda vráti informáciu, či je zásobník prázdny.
MF_Stack2	Push	Metóda pridá prvok na vrchol zásobníku.
	Pop	Metóda odoberie prvok z vrchu zásobníku.
	Clear	Vymazanie prvkov zásobníku mimo počet predaný, ako parameter tejto metódy.
	ClearAll	Úplná finalizácia objektu.
	GetSize	Metóda vráti veľkosť poľa.
	GetCountOfUsedMemBlocks	Metóda vráti počet pamäťových blokov, ktoré sú vo vlastníctve objektu dát.štruktúry.
	GetSignBlockSize	Metóda vráti príznak veľkosti použitých pamäťových blokov objektom dát. štruktúry: S, B, T.
	IsEmpty	Metóda vráti informáciu, či je zásobník prázdny.
MF_List	Add	Vloženie prvku na koniec zoznamu
	GetHeadNodeData	Prístup dátam k prvého prvku zoznamu.
	GetTailNodeData	Prístup k dátam posledného prvku zoznamu
	GetNodesData	Prístup k prvku, ktorého pozícia je predaná parametrom
	EraseOnPosition	Odstránenie prvku (prvkov) na danej pozicií. Pri väčšom počte prvkov je parametrom rozsah.
	Erase	Odstránenie všetkých prvkov zo zoznamu s dátami zadanými ako parameter metódy.
	Clear	Vymazanie prvkov zoznamu mimo počet predaný, ako parameter tejto metódy.
	ClearAll	Úplná finalizácia objektu.
	IsEmpty	Metóda vráti informáciu, či je zoznam prázdny.
	GetCountOfUsedMemBlocks	Metóda vráti počet pamäťových blokov, ktoré sú vo vlastníctve objektu dát.štruktúry.
	GetSignBlockSize	Metóda vráti príznak veľkosti použitých pamäťových blokov objektom dát. štruktúry: S, B, T.

B Príloha – Triedne diagramy

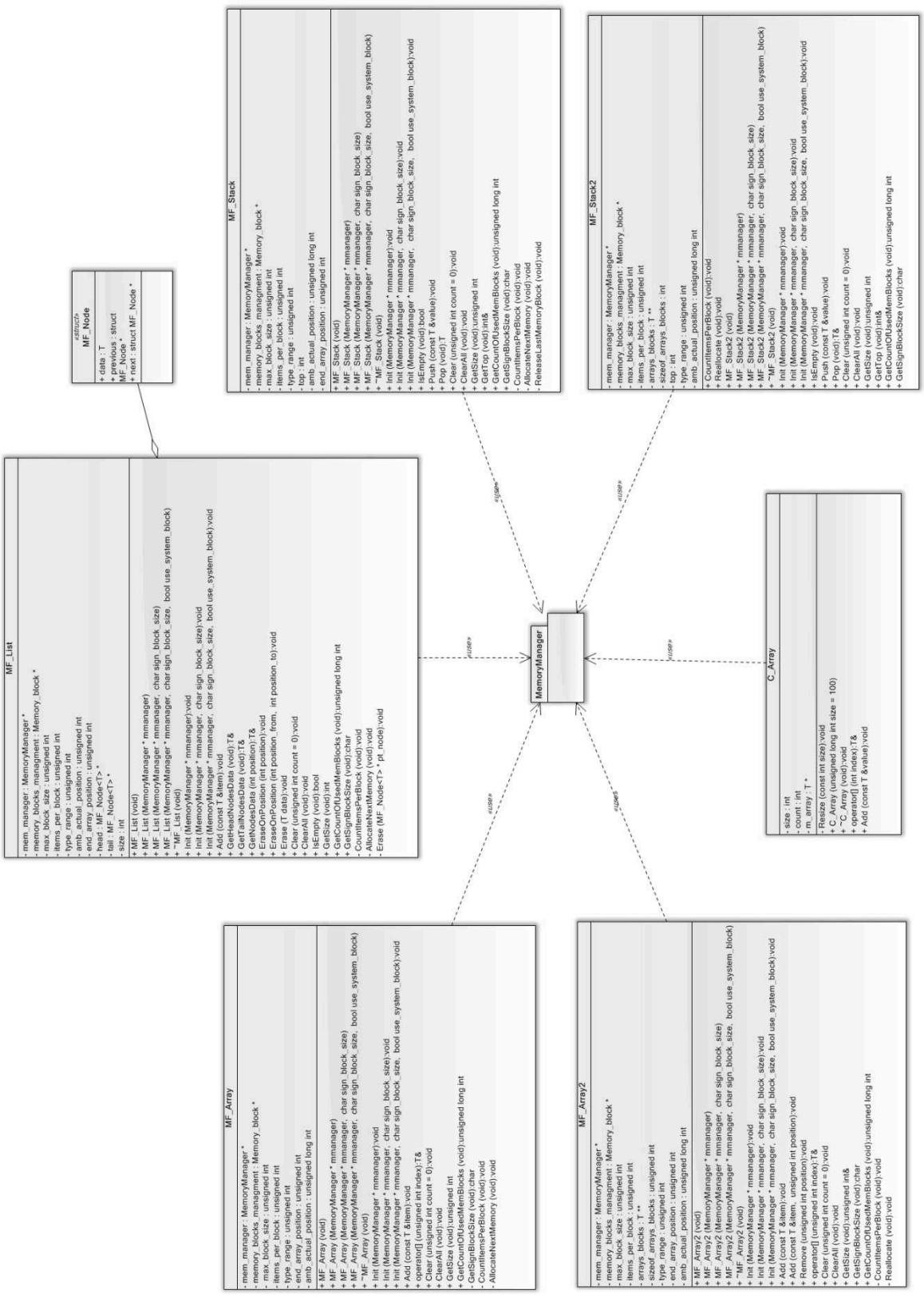
B.1 Diagram štruktúry balíčkov



B.2 Diagram tried balíčku (modulu) správy pamäte



B.3 Diagram tried balíčku (modulu) dátových štruktúr

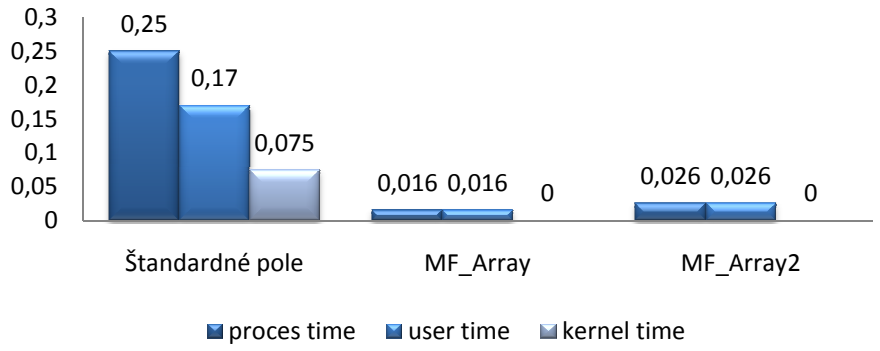


C Príloha – Výsledky testov

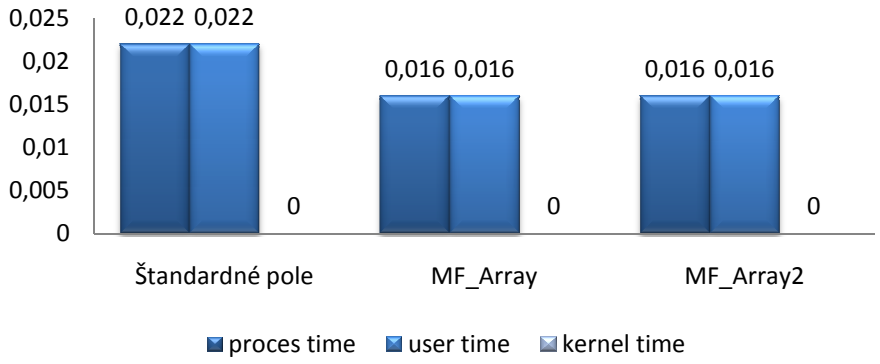
C.1 Výsledky 1. výkonnostného testu

Test rýchlosti alokácie a dealokácie väčšieho množstva polí jednotlivých typov. Celkom sme alokovali 30 000 polí o veľkosti 170 prvkov. Dátové štruktúry rámca boli postavené na bloku veľkosti 8192B.

Alokácia polí								
Štandardné pole			MF_Array			MF_Array2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
0,250	0,170	0,075	0,016	0,016	0	0,026	0,026	0



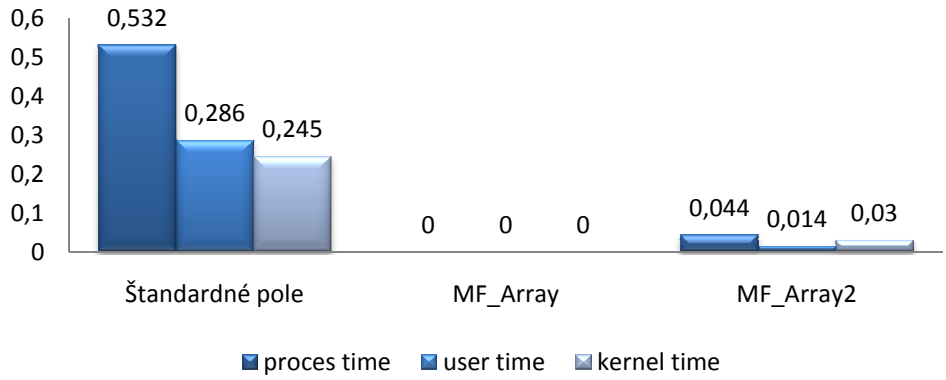
Dealokácia polí								
Štandardné pole			MF_Array			MF_Array2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
0,022	0,022	0	0,012	0,012	0	0,016	0,016	0



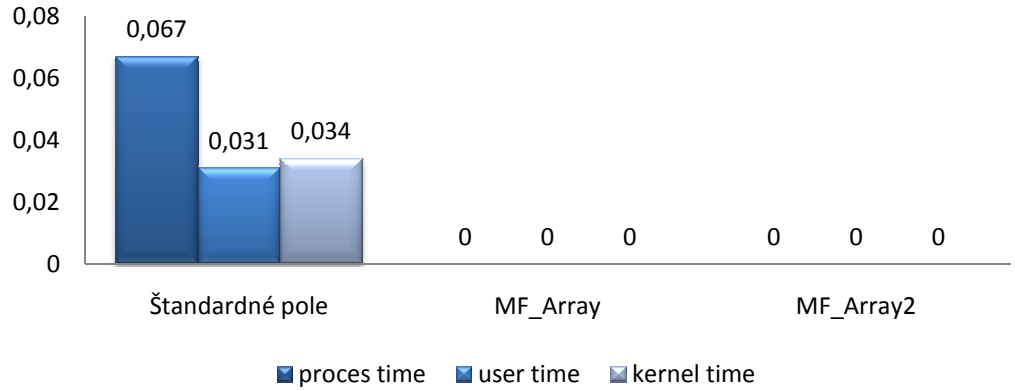
Spotreba pamäte					
Štandardné pole (new/delete)		MF_Array		MF_Array2	
Spotrebovaná pam.:	244 884 KB	Spotrebovaná pam.:	240 000 KB	Spotrebovaná pam.:	240 000 KB
Uvolnená pamäť:	4 KB	Uvolnená pamäť:	240 000 KB	Uvolnená pamäť:	240 000 KB
Rozdiel:	-244 880 KB	Rozdiel:	0 KB	Rozdiel:	0 KB

V alternatívnom teste sme alokovali a dealokovali len 10 polí každého typu, pričom pole mali mnohonásobne väčšiu veľkosť (až 48 000 000B).

Alokácia polí								
Štandardné pole			MF_Array			MF_Array2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
0,532	0,286	0,245	0	0	0	0,044	0,014	0,03



Dealokácia polí								
Štandardné pole			MF_Array			MF_Array2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
0,022	0,022	0	0,012	0,012	0	0,016	0,016	0

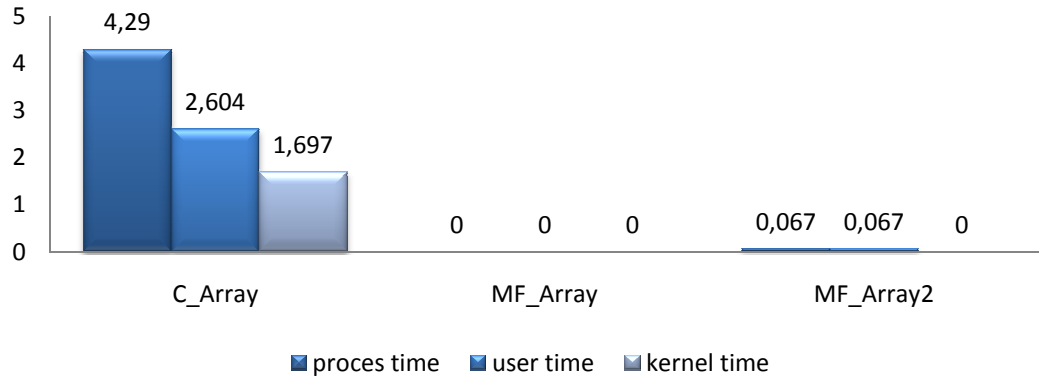


Spotreba pamäte					
Štandardné pole (new/delete)		MF_Array		MF_Array2	
Spotrebovaná pam.:	470 040 KB	Spotrebovaná pam.:	468 750 KB	Spotrebovaná pam.:	468 750 KB
Uvolnená pamäť:	469 680 KB	Uvolnená pamäť:	468 750 KB	Uvolnená pamäť:	468 750 KB
Rozdiel:	-360 KB	Rozdiel:	0 KB	Rozdiel:	0 KB

C.2 Výsledky 2. výkonnostného testu

Test rýchlosti vloženia 100 000 prvkov do jednotlivých typov polí a rýchlosti náhodného prístupu k 100 000 000 prvkov.

Doba vloženie prvkov								
Štandardné pole			MF_Array			MF_Array2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
4,290	2,604	1,697	0	0	0	0,067	0,067	0



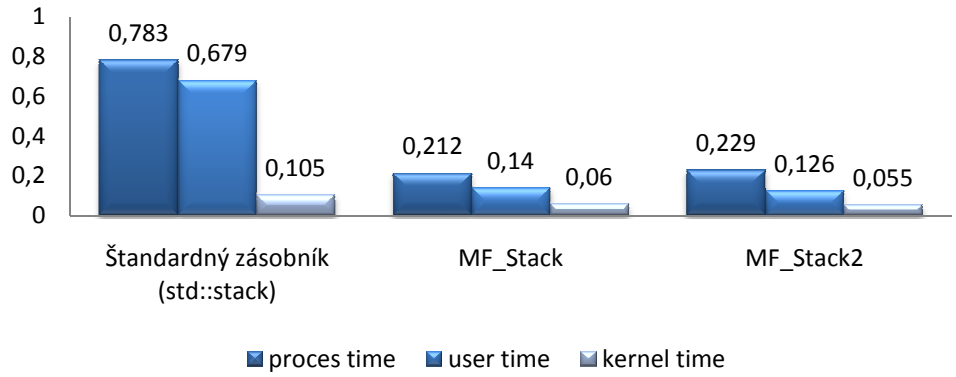
Doba prístupu k prvkov								
Štandardné pole			MF_Array			MF_Array2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
0	0	0	0	0	0	0	0	0

Spotreba pamäte					
C_Array		MF_Array		MF_Array2	
Spotrebovaná pam.:	6836 KB	Spotrebovaná pam.:	4736 KB	Spotrebovaná pam.:	4849 KB

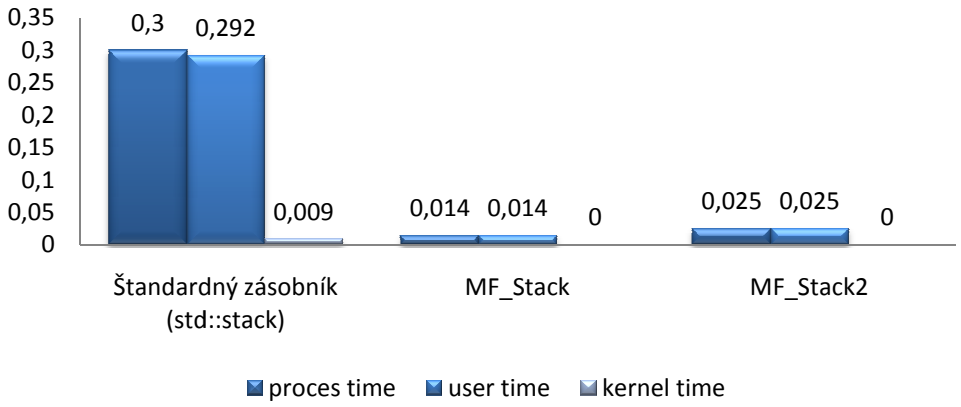
C.3 Výsledky 3. výkonnostného testu

Test rýchlosti alokácie a dealokácie väčšieho množstva zásobníkov u jednotlivých typov. Celkom bolo vyalokovaných 30 000 zásobníkov každého typu a do každého zásobníku bolo vložených 150 prvkov veľkosti 48B.

Alokácia zásobníkov								
Štandardný zásobník (std::stack)			MF_Stack			MF_Stack2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
0,783	0,679	0,105	0,212	0,014	0,06	0,229	0,126	0,055



Dealokácia zásobníkov								
Štandardný zásobník (std::stack)			MF_Stack			MF_Stack2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
0,3	0,292	0,009	0,014	0,014	0	0,025	0,025	0

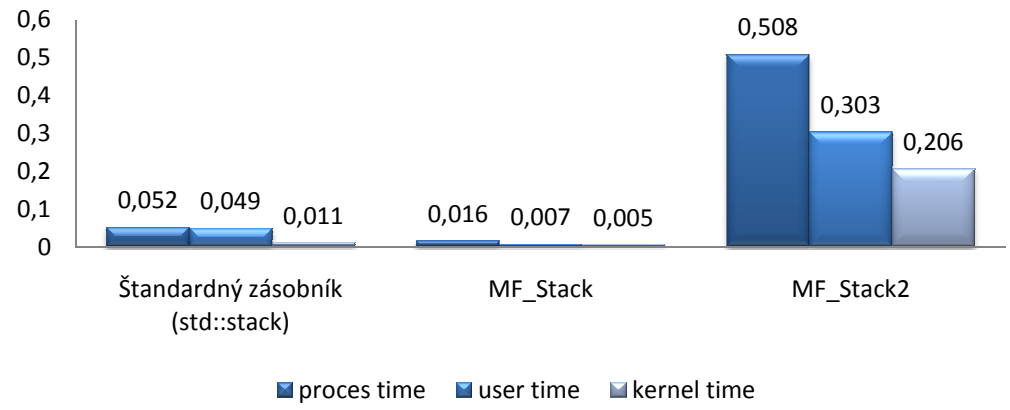


Spotreba pamäte					
Štandardný zásobník (std::stack)		MF_Stack		MF_Stack2	
Spotrebovaná pam.:	271 112 KB	Spotrebovaná pamäť:	240 000 KB	Spotrebovaná pam.:	240 000 KB
Uvolnená pamäť:	114 372 KB	Uvolnená pamäť:	240 000 KB	Uvolnená pamäť:	240 000 KB
Rozdiel:	-156 740 KB	Rozdiel:	0 KB	Rozdiel:	0 KB

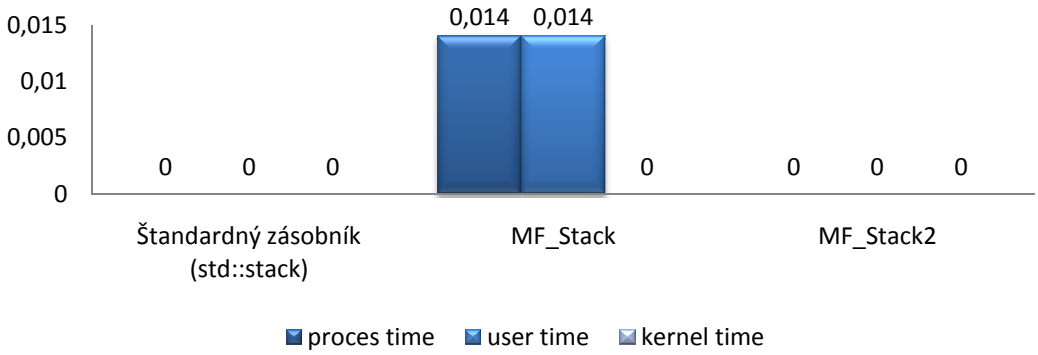
C.4 Výsledky 4.výkonnostného testu

Test rýchlosti vloženia (push) a výberu (pop) 200 000 prvkov typu TestObject (veľkosť objektu 48B) u jednotlivých typov zásobníkov.

Vloženie (push) prvkov do zásobníkov								
Štandardný zásobník (std::stack)			MF_Stack			MF_Stack2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
0,052	0,049	0,011	0,016	0,007	0,005	0,508	0,303	0,206



Výber (pop) prvkov zo zásobníkov								
Štandardný zásobník (std::stack)			MF_Stack			MF_Stack2		
process time	user time	kernel time	process time	user time	kernel time	process time	user time	kernel time
0	0	0	0,014	0,014	0	0	0	0

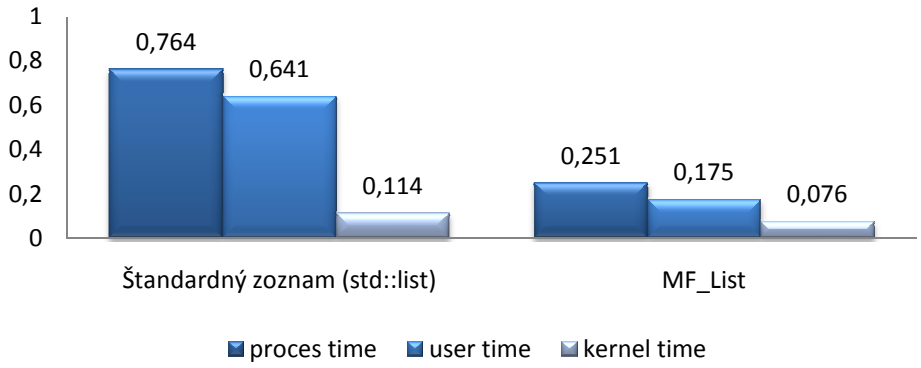


Spotreba pamäte					
Štandardný zásobník (std::stack)		MF_Stack		MF_Stack2	
Spotrebovaná pamäť:	12 636 KB	Spotrebovaná pamäť:	9472 KB	Spotrebovaná pamäť:	9472 KB
Uvolnená pamäť:	0 KB	Uvolnená pamäť:	9472 KB	Uvolnená pamäť:	0 KB
Rozdiel:	-12 636 KB	Rozdiel:	0 KB	Rozdiel:	9472 KB

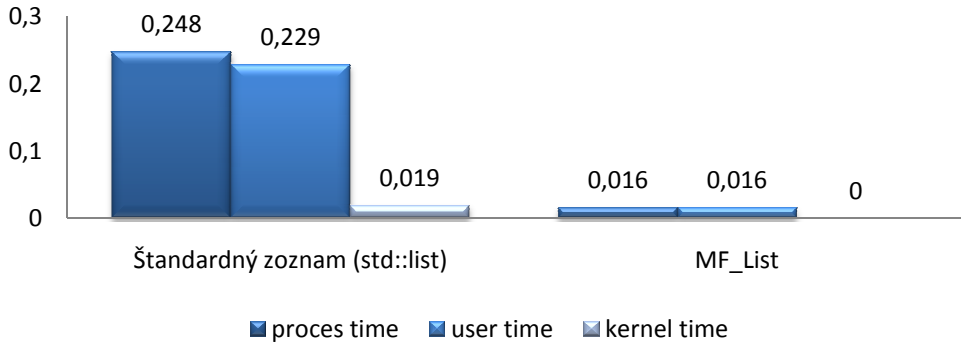
C.5 Výsledky 5.výkonnostného testu

Test rýchlosti alokácie a dealokácie väčšieho množstva zoznamov u jednotlivých typov. Celkom bolo vyalokovaných 30 000 zoznamov každého typu a do každého zoznamu bolo vložených 150 prvkov veľkosti 48B.

Alokácia zoznamov					
Štandardný zoznam (std::list)			MF_List		
process time	user time	kernel time	process time	user time	kernel time
0,764	0,641	0,114	0,251	0,175	0,076



Dealokácia zoznamov					
Štandardný zoznam (std::list)			MF_List		
process time	user time	kernel time	process time	user time	kernel time
0,248	0,229	0,019	0,016	0,016	0



Spotreba pamäte			
Štandardný zásobník (std::list)		MF_List	
Spotrebovaná pamäť:	285 088 KB	Spotrebovaná pamäť:	480 000 KB
Uvolnená pamäť:	156 932 KB	Uvolnená pamäť:	480 000 KB
Rozdiel:	-12 636 KB	Rozdiel:	0 KB

C.6 Nezávislí test alokácie a uvolňovania štandardných polí

Nezávislím testom alokácie a uvolňovania štandardných polí je test, ktorý sme previedli za účelom simulácie problému s neumplným uvolňovaním pamäte u väčšieho množstva malých pamäťových blokov pod 512 KB. Test bol implementovaný, ako samostatná aplikácia, aby sa zamedzilo prípadnému úniku pamäte iným procesom. Test obsahuje dve metódy, kde prvá metóda alokuje a dealokuje jednorozmerné pole typu `Integer` veľkosti *(512*512*512)* a druhá metóda alokuje a dealokuje dvojrozmerné pole typu Integer, kde prvý rozmer má veľkosť *(512*512)* a druhý rozmer veľkosť 512. Celkovo obidve polia alokujú pamäť veľkosti cca 500MB.

Test bol prevedený na dvoch PC s operačnými systémami MS Windows Vista Home Premium 32-bit a MS Windows 7 Professional 32-bit.

Výsledkom testu je stav pamäte zaznamenaný pomocou utility *VMmap v3.1* v priebehu jednotlivých alokácií a dealokácií jednorozmerného a dvojrozmerného poľa.

	MS Windows Vista Home Premium, 32-bit	**MS Windows 7 Professional, 32-bit**
Jednorozmerné pole		
Stav pamäte pred alokáciou	1416 KB	1496 KB
Stav pamäte po alokácii	526 740 KB	526 820 KB
Stav pamäte po dealokácii	1424 KB	1504 KB
Dvojrozmerné pole		
Stav pamäte pred alokáciou	1424 KB	1504 KB
Stav pamäte po alokácii	532 130 KB	532 212 KB
Stav pamäte po dealokácii	47 220 KB	4 904 KB